Polynomial-time what-if analysis for communication networks: An automata-theoretic approach

Stefan Schmid *et al.**

* most importantly: Jiri Srba (Aalborg University) and
Chen Avin (Ben Gurion University)

Nice to meet you!

Polynomial-time what-if analysis for communication networks: An automata-theoretic approach

Stefan Schmid *et al.**

* most importantly: Jiri Srba (Aalborg University) and Chen Avin (Ben Gurion University)

Nice to meet you!

Polynomial-time what-if analysis for communication networks: An automata-theoretic approach

Patent pending, INFOCOM 2018

Stefan Schmid *et al.**

New in Austria, looking for collaborations etc.

G.I.F. project

* most importantly: Jiří Srba (Aalborg University) and Chen Avin (Ben Gurion University)

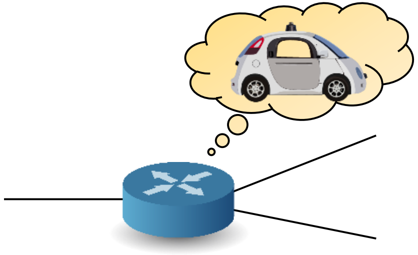
Polynomial-time what-if analysis for communication networks: An automata-theoretic approach

Stefan Schmid *et al.**

Formal methods are *the* hot topic in networking! And your expertise may be one of the most urgently required ones…

Communication Technologies (CT) @ Uni Vie

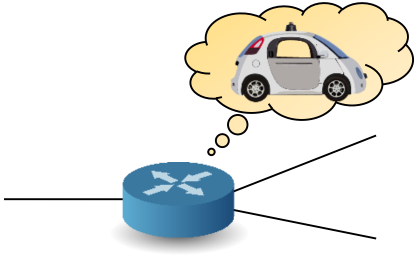
- Vision and mission: Make networked systems **self-***
  - Self-repairing
  - Self-stabilizing
  - Self-adjusting
  - Self-”driving”



- Using different methodologies
  - **Algorithms** and analysis (LPs, online/approx. algorithms, etc.)
  - **Formal methods** (e.g., automata theory and synthesis)
  - **Machine-learning** (e.g., data-driven optimizations)

Communication Technologies (CT) @ Uni Vie

- Vision and mission: Make networked systems **self-***
  - Self-repairing
  - Self-stabilizing
  - Self-adjusting
  - Self-”driving”

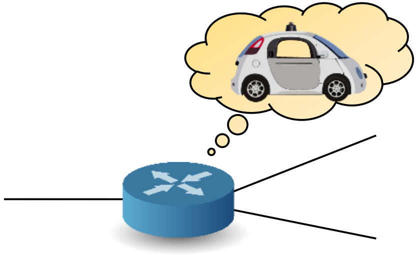


- Using different methodologies
  - **Algorithms** and analysis (LPs, online/approx. algorithms, etc.)
  - **Formal methods** (e.g., automata theory and synthesis)
  - **Machine-learning** (e.g., data-driven optimizations)

Uwe Nestmann (CONCUR 2016)

Communication Technologies (CT) @ Uni Vie

- Vision and mission: Make networked systems **self-***
  - Self-repairing
  - Self-stabilizing
  - Self-adjusting
  - Self-”driving”



- Using different met ies
  - **Algorithm** online/approx. algorithms, etc.)
  - **For** , automata theory and synthesis)
  - **Mac ing** (e.g., data-driven optimizations)

Ideally: From practice to theory and back!

Why Self-*? Complexity and Human Errors!

Datacenter, enterprise, carrier networks: **mission-critical infrastructures**.
But even **techsavvy** companies struggle to provide reliable operations.

We discovered a misconfiguration on this pair of switches that caused what's called a "bridge loop" in the network.

A network change was […] executed incorrectly […] more "stuck" volumes and added more requests to the re-mirroring storm.

Service outage was due to a series of internal network events that corrupted router data tables.

Experienced a network connectivity issue […] interrupted the airline's flight departures, airport processing and reservations systems

Credits: Nate Foster

Why Self-*? Lack of Good Debugging Tools!

The **Wall Street bank** anecdote*: **datacenter outage*** of a Wall Street investment bank led to revenue loss measured in ***USD 10^6 / min***!

Quickly, assembled **emergency team**:

The compute team: quickly came armed with ***reams of logs***, showing ***how*** and when the applications failed, and had already ***written experiments*** to reproduce and isolate the error, along with candidate prototype programs to workaround the failure.

The storage team: similarly equipped, showing which file ***system logs*** were affected, and already progressing ***with workaround programs***.

The networking team: All the networking team had were ***two tools invented over twenty years ago*** [*ping* and *traceroute*] to merely ***test end-to-end connectivity***. Neither tool could reveal problems with the ***switches***, the ***congestion*** experienced by individual packets, or provide any means to create experiments to identify, quarantine and resolve the problem.

Source: «The world’s fastest and most programmable networks»
White Paper Barefoot Networks

Why Self-*? Lack of Good Deb ools!

Who was blamed?

The **Wall Street bank** anecdote*: **datacenter outage*** of a Wall Street investment bank led to revenue loss measured in ***USD 10^6 / min***!

Quickly, assembled **emergency team**:

The compute team: quickly came armed with ***reams of logs***, showing ***how*** and when the applications failed, and had already ***written experiments*** to reproduce and isolate the error, along with candidate prototype programs to workaround the failure.

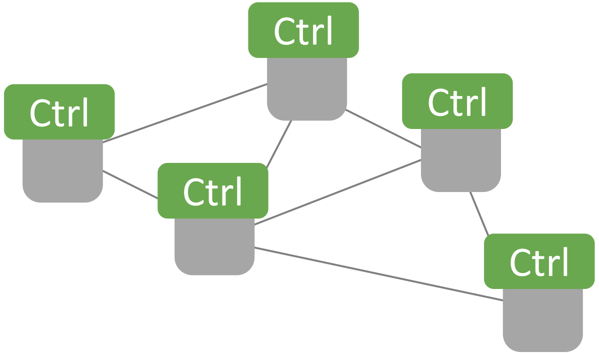
The storage team: similarly equipped, showing which file ***system logs*** were affected, and already progressing ***with workaround programs***.

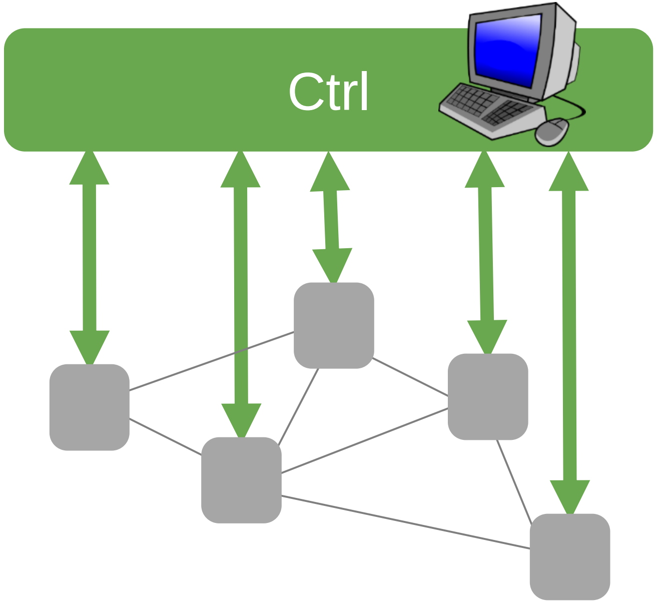
The networking team: All the networking team had were ***two tools invented over twenty years ago*** [*ping* and *traceroute*] to merely ***test end-to-end connectivity***. Neither tool could reveal problems with the ***switches***, the ***congestion*** experienced by individual packets, or provide any means to create experiments to identify, quarantine and resolve the problem.

Source: «The world’s fastest and most programmable networks»
White Paper Barefoot Networks

Why Self-*? Flexibility!

Communication networks are becoming more **flexible (general)** and **software-defined**.



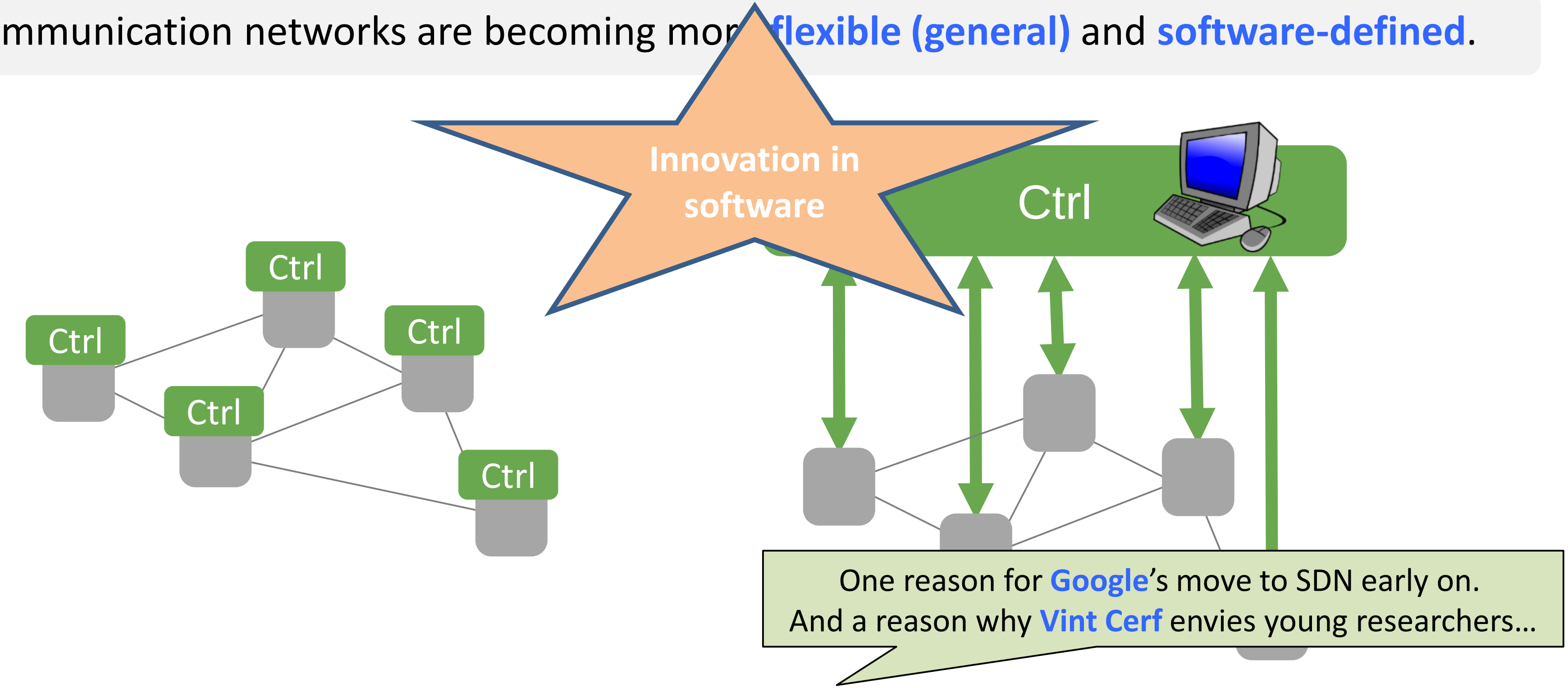


Traditional networks: ***distributed*** and ***fixed*** algorithms and functionality, ***blackbox***

Software-Defined Networks (SDNs): centralized control, ***bring-your-own-algorithm***, passive match-action rules (***verifiable***)

Why Self-*? Flexibility!

Communication networks are becoming more **flexible (general)** and **software-defined**.



One reason for **Google**'s move to SDN early on.
And a reason why **Vint Cerf** envies young researchers…

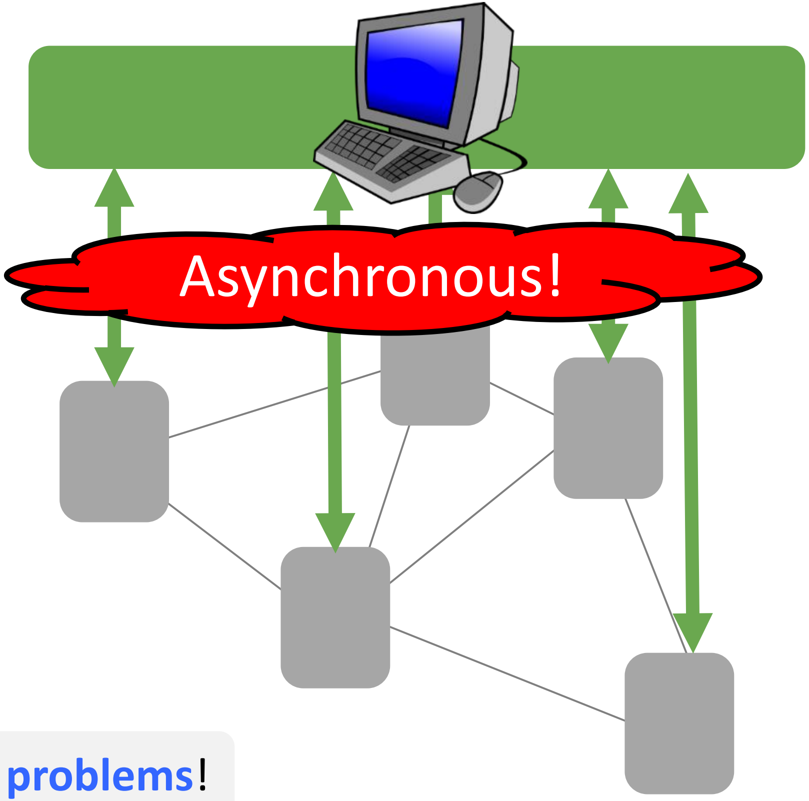
Traditional networks: ***distributed*** and ***fixed*** algorithms and functionality, ***blackbox***

Software-Defined Networks (SDNs): centralized control, ***bring-your-own-algorithm***, passive match-action rules (***verifiable***)

Software-Defined Networks (SDNs)

- Networks become **programmable**, **open** and **more general**
  - "the Linux of networking"
  - Support **expressive forwarding**: match-action on Layer-2 to Layer-4
  - Programmatic, **adaptive** control

- But also introduces ***new challenges***:
  - More general = harder?
  - E.g., **decoupling** (remote controller)



Exploiting flexibilities introduces novel **algorithmic problems**!

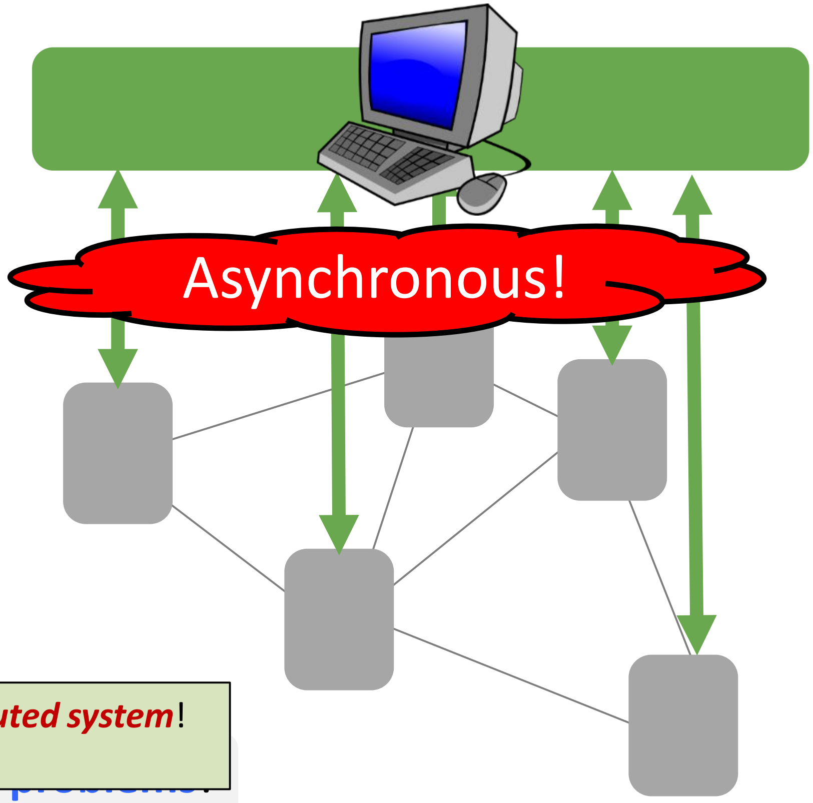
Software-Defined Networks (SDNs)

- Networks become **programmable**, **open** and **more general**
  - “the Linux of networking”
  - Support **expressive forwarding**: match-action on Layer-2 to Layer-4
  - Programmatic, **adaptive** control

- But also introduces ***new challenges***:
  - More general = harder?
  - E.g., **decoupling** (remote controller)

Gives raise to non-trivial **inconsistencies**: A ***distributed system***!
And case for automated ***verification***…

Exploitin



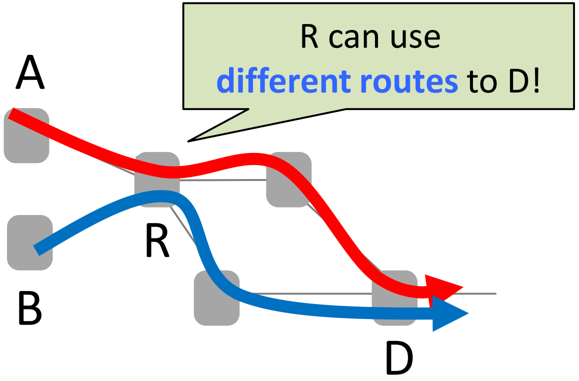
What’s new? Examples.

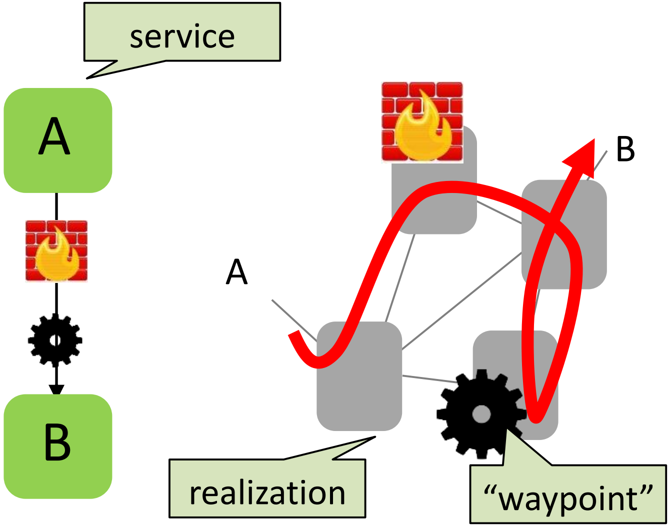
- Traditional **traffic engineering**: routes can only be influenced ***indirectly***, using ***link weights*** as knobs, only ***shortest paths***

- **SDN**: ***direct control*** over forwarding rules and hence routing paths

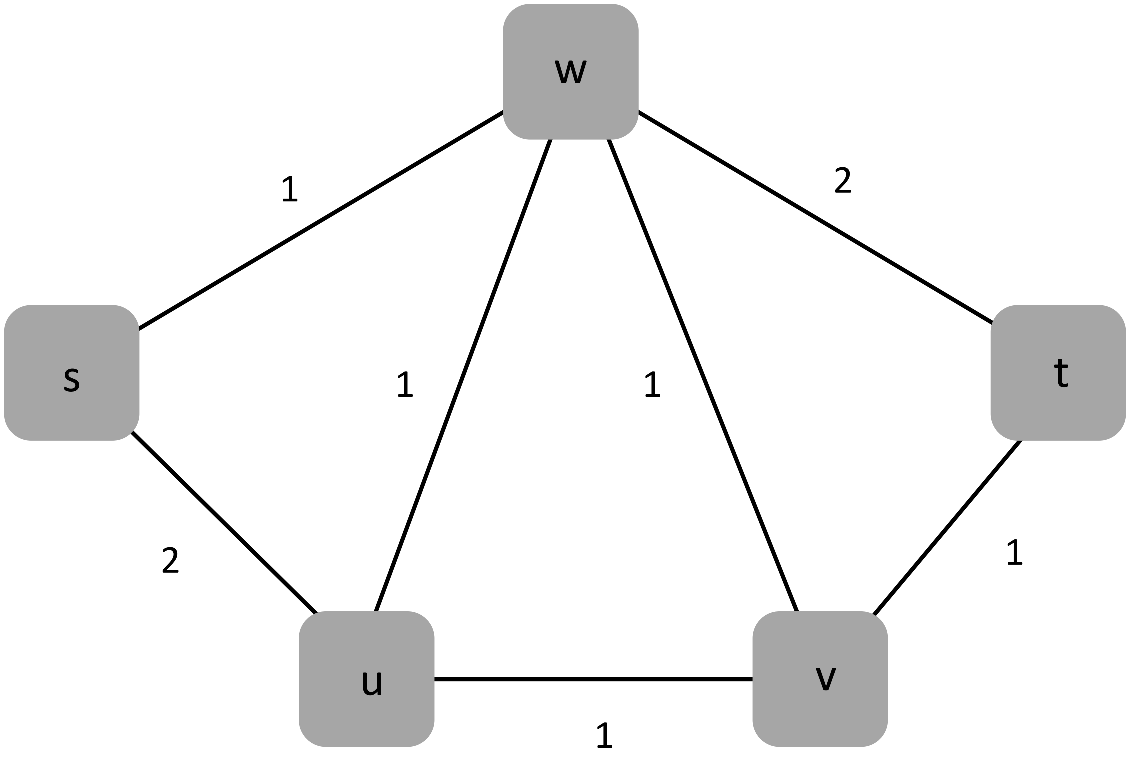
- Routes also do not have to be destination-based or ***confluent*** (but can depend on other ***header fields***)

- Routes may even contain ***loops*** (not a simple path but a **walk**): steered through network functions to provide complex network service (**service chain**)

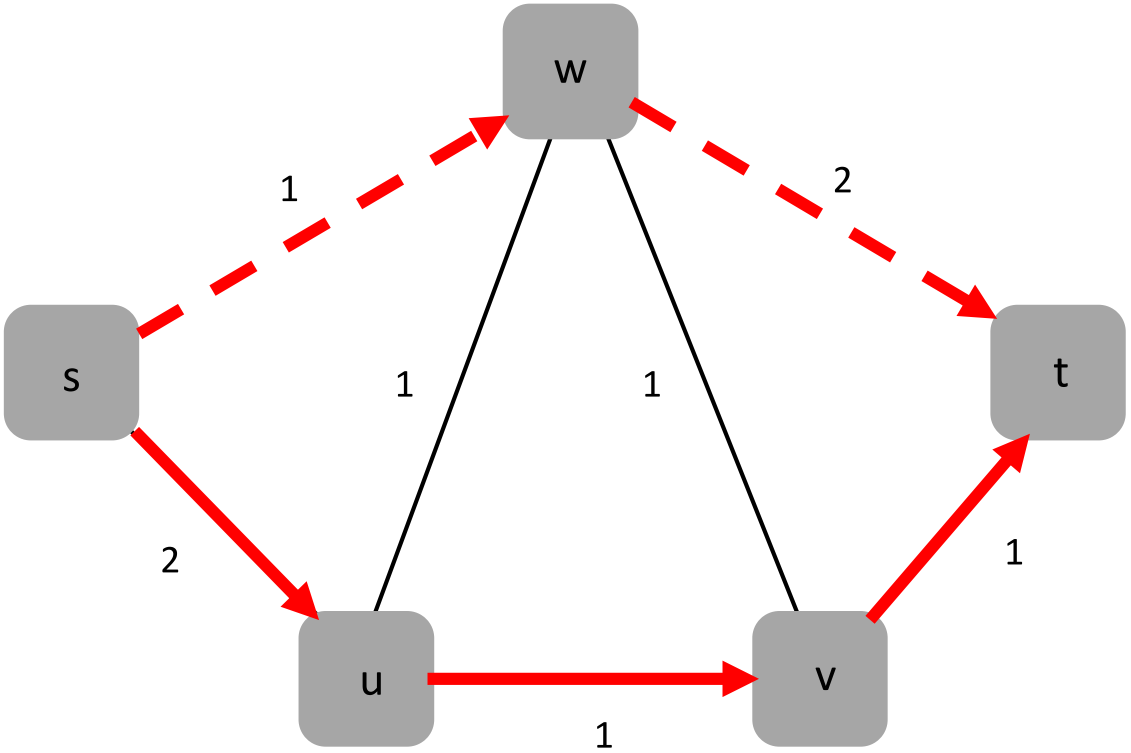




Example: Consistent Network Updates

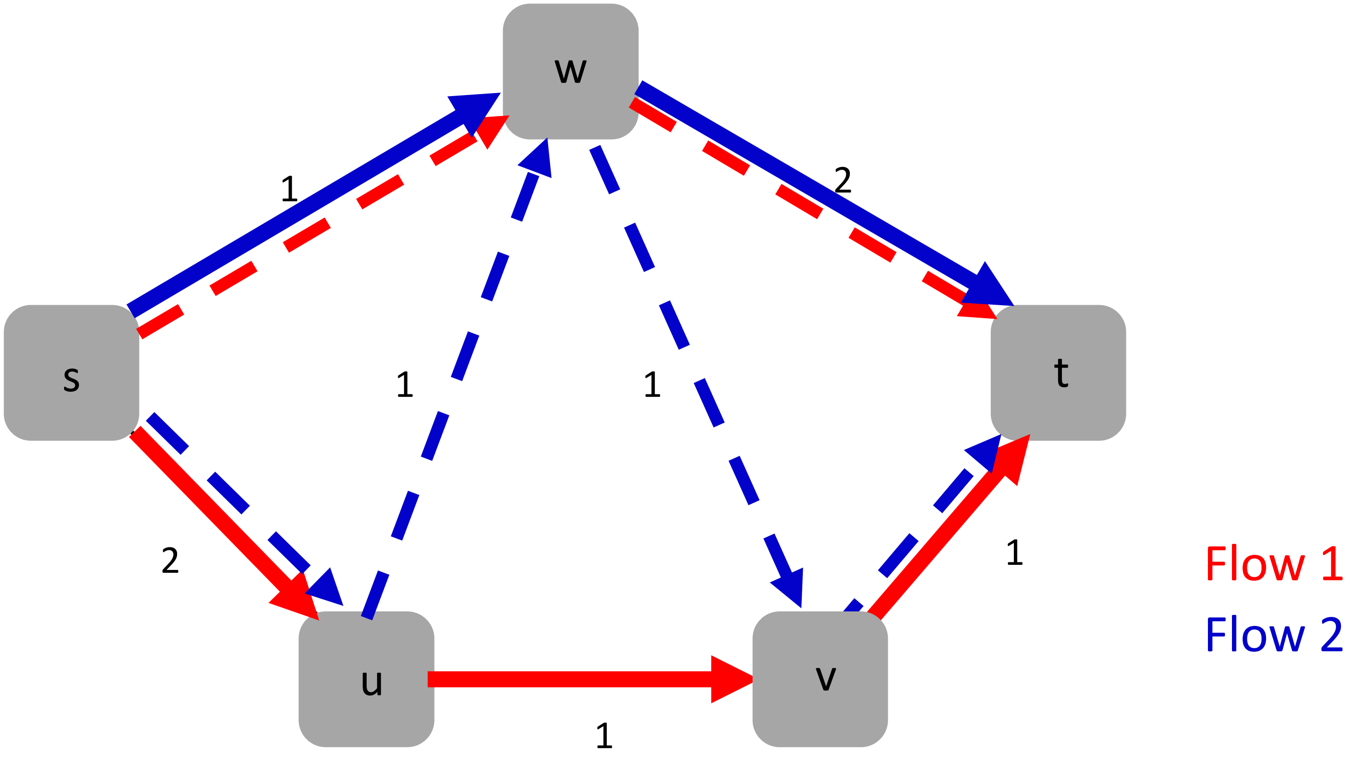


Example: Consistent Network Updates

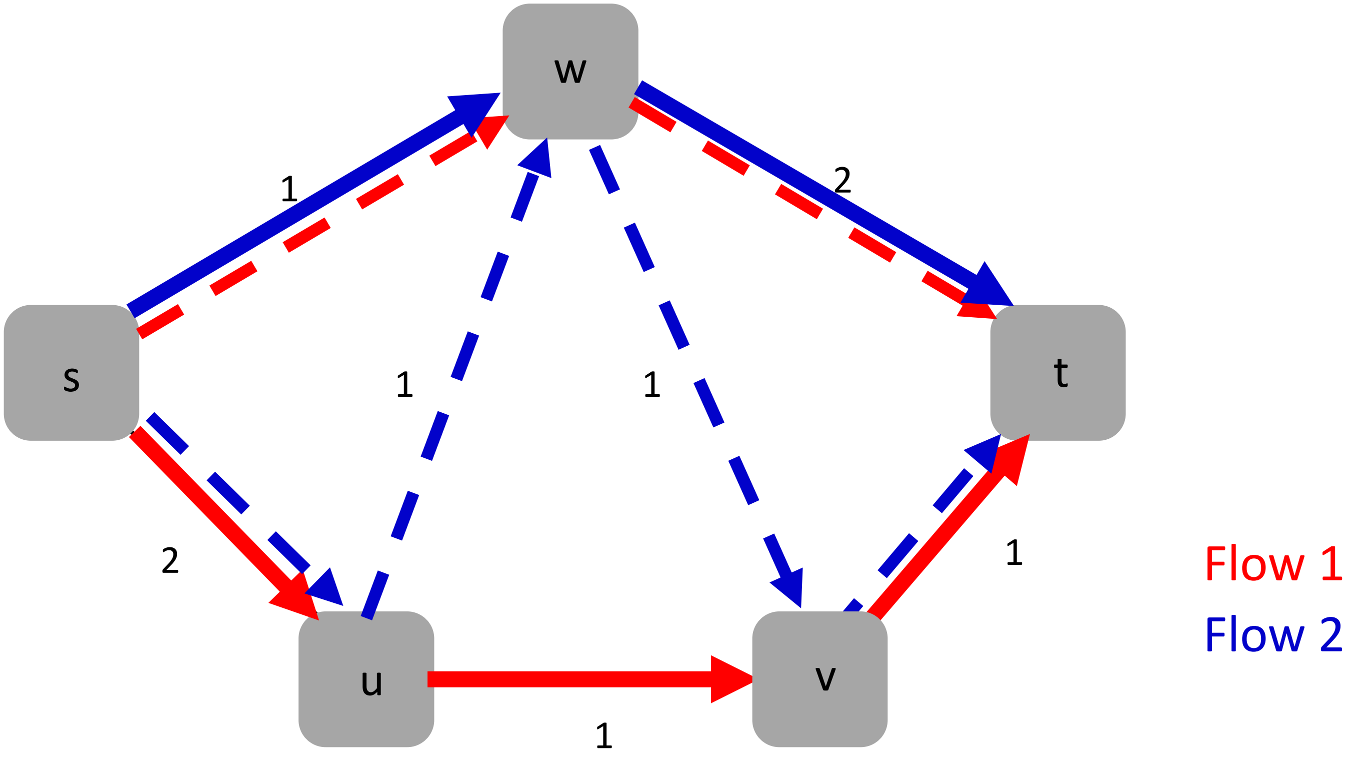


Flow 1

Example: Consistent Network Updates



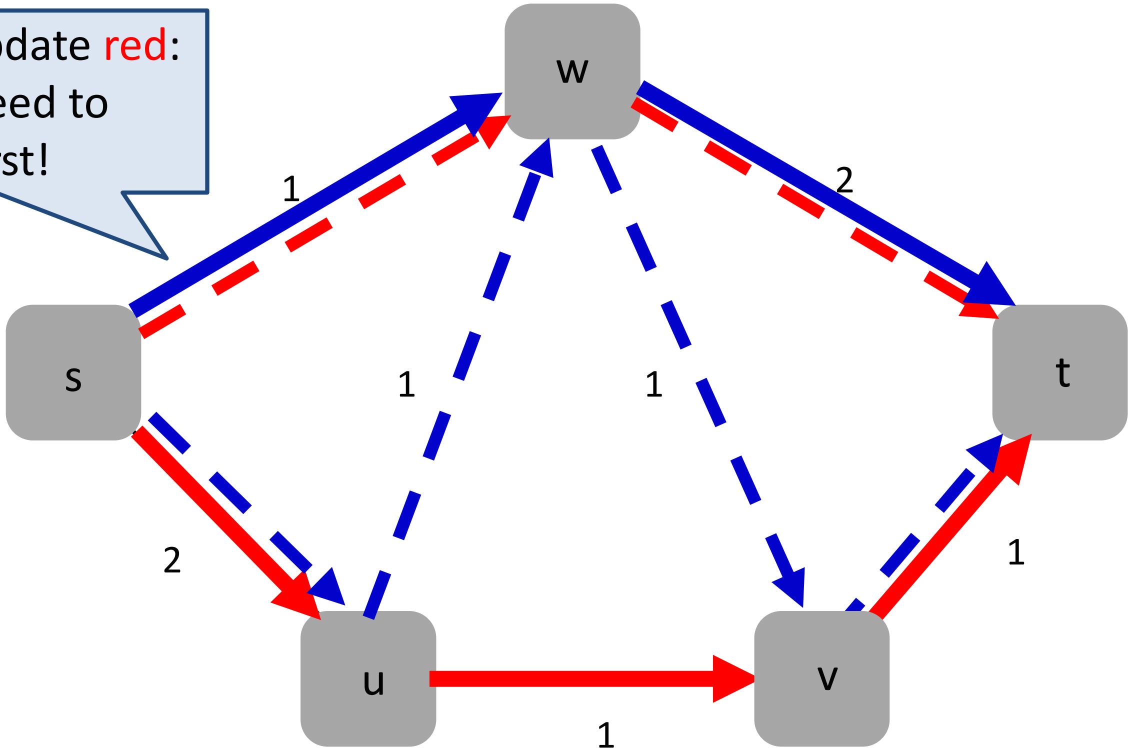
Example: Consistent Network Updates



(Short) congestion-free update schedule?

Example: Consistent Network Updates

e.g., cannot update red: congestion! Need to update blue first!



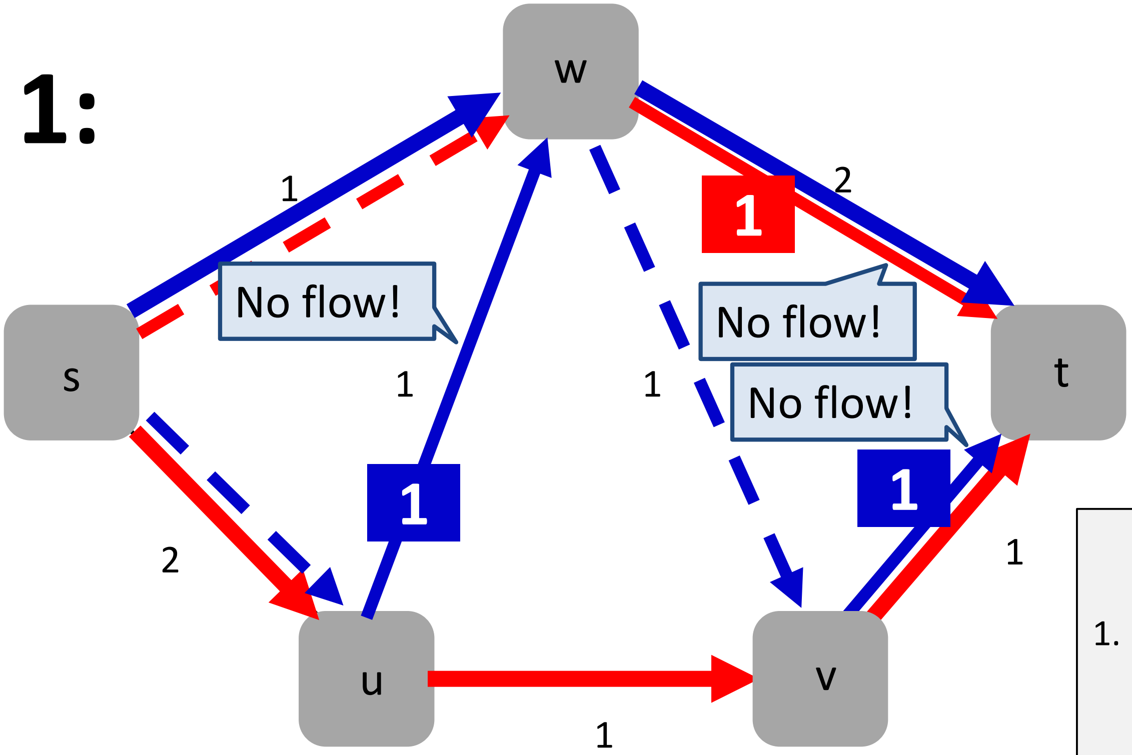
Flow 1

Flow 2

(Short) congestion-free update schedule?

Example: Consistent Network Updates

Round 1:



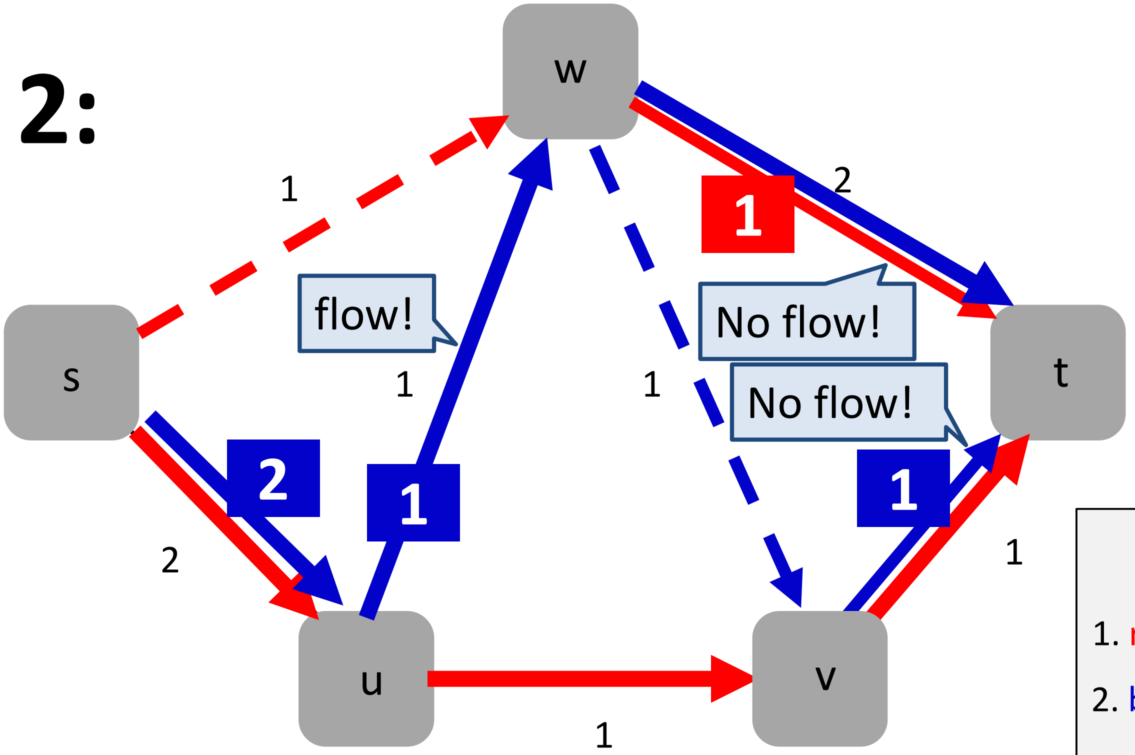
Prepare!

Schedule:

1. red@w,blue@u,blue@v

Example: Consistent Network Updates

Round 2:



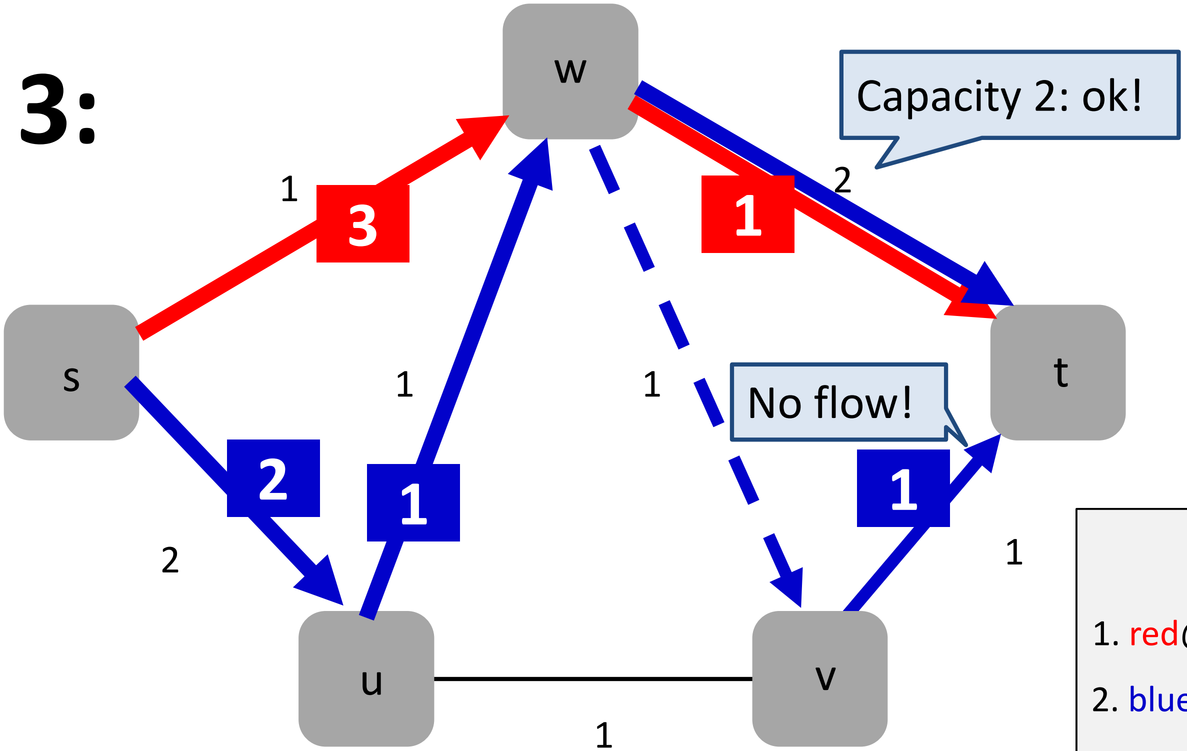
Schedule:

1. red@w,blue@u,blue@v

2. blue@s

Example: Consistent Network Updates

Round 3:



Capacity 2: ok!

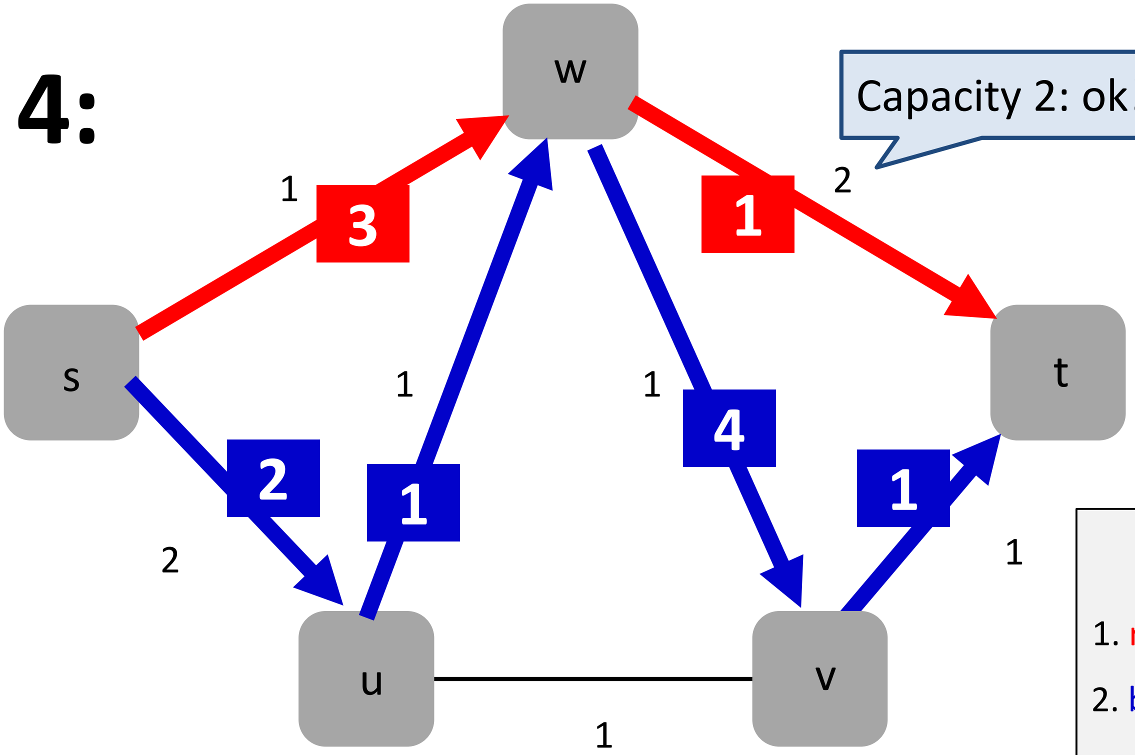
No flow!

Schedule:

1. red@w,blue@u,blue@v
2. blue@s
3. red@s

Example: Consistent Network Updates

Round 4:



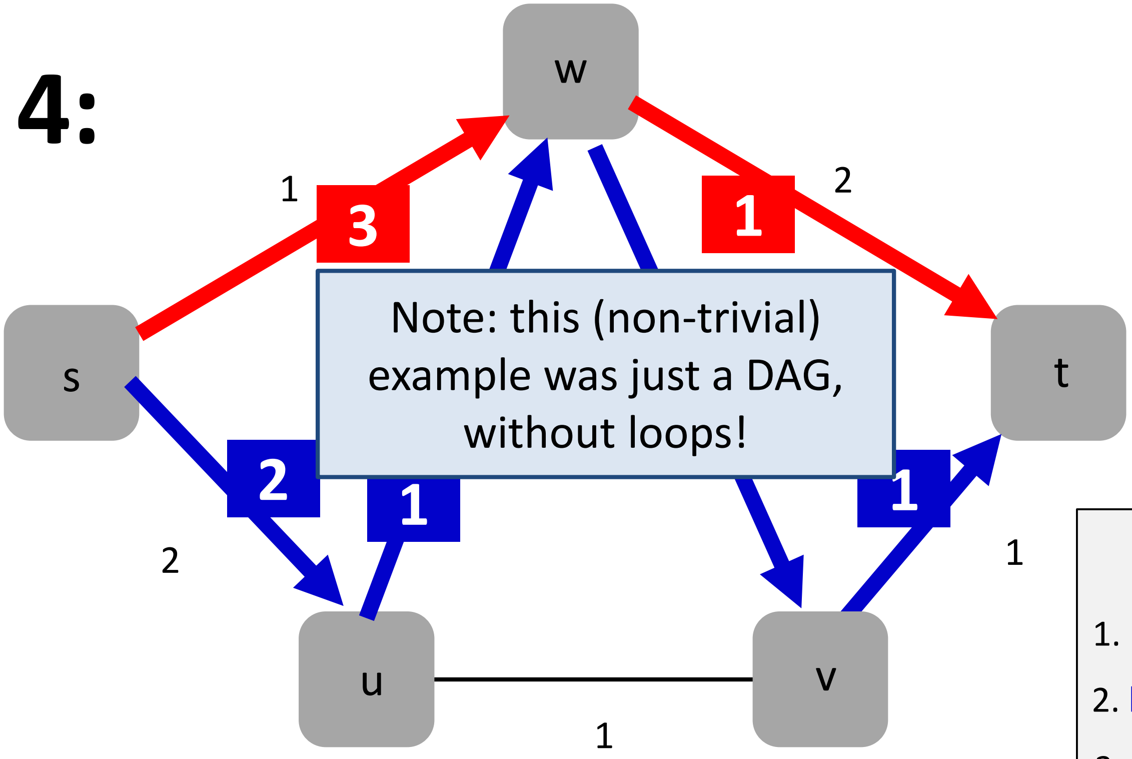
Capacity 2: ok!

Schedule:

1. red@w,blue@u,blue@v
2. blue@s
3. red@s
4. blue@w

Example: Consistent Network Updates

Round 4:

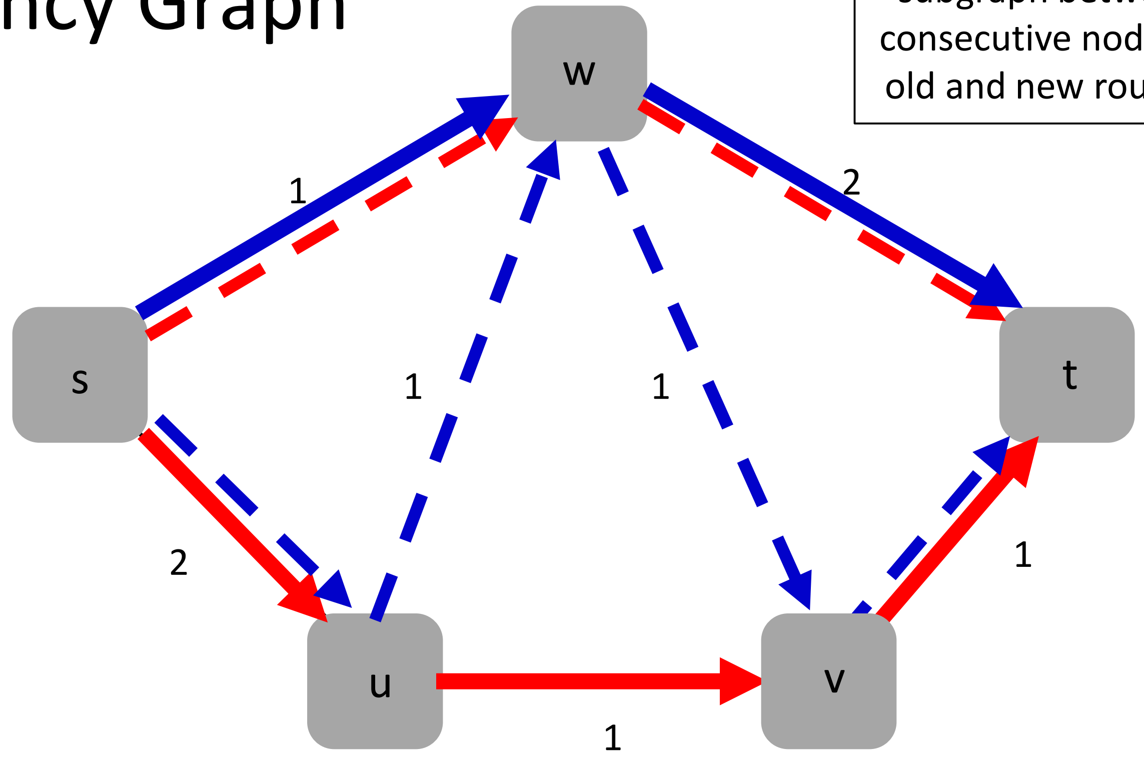


Note: this (non-trivial) example was just a DAG, without loops!

Schedule:

1. red@w,blue@u,blue@v
2. blue@s
3. red@s
4. blue@w

Block Decomposition and Dependency Graph



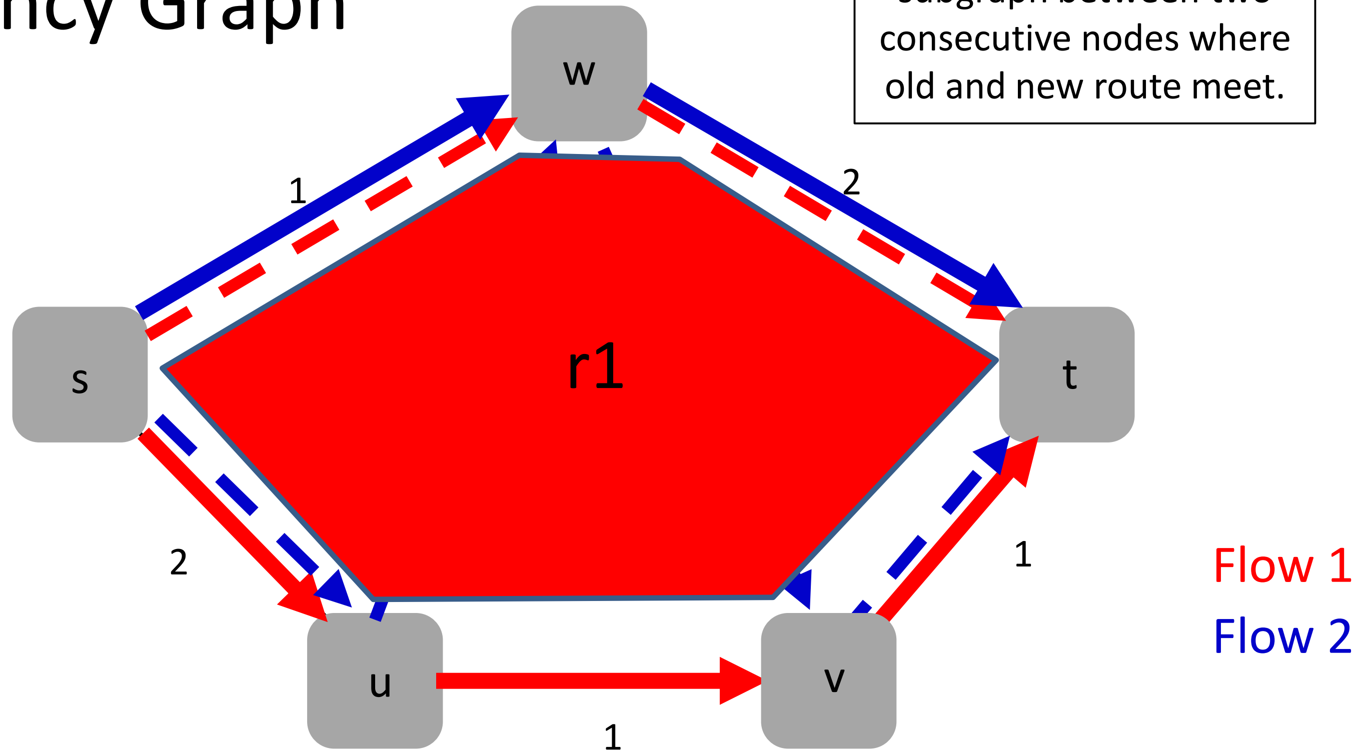
Block for a given flow: subgraph between two consecutive nodes where old and new route meet.

Flow 1

Flow 2

Block Decomposition and Dependency Graph

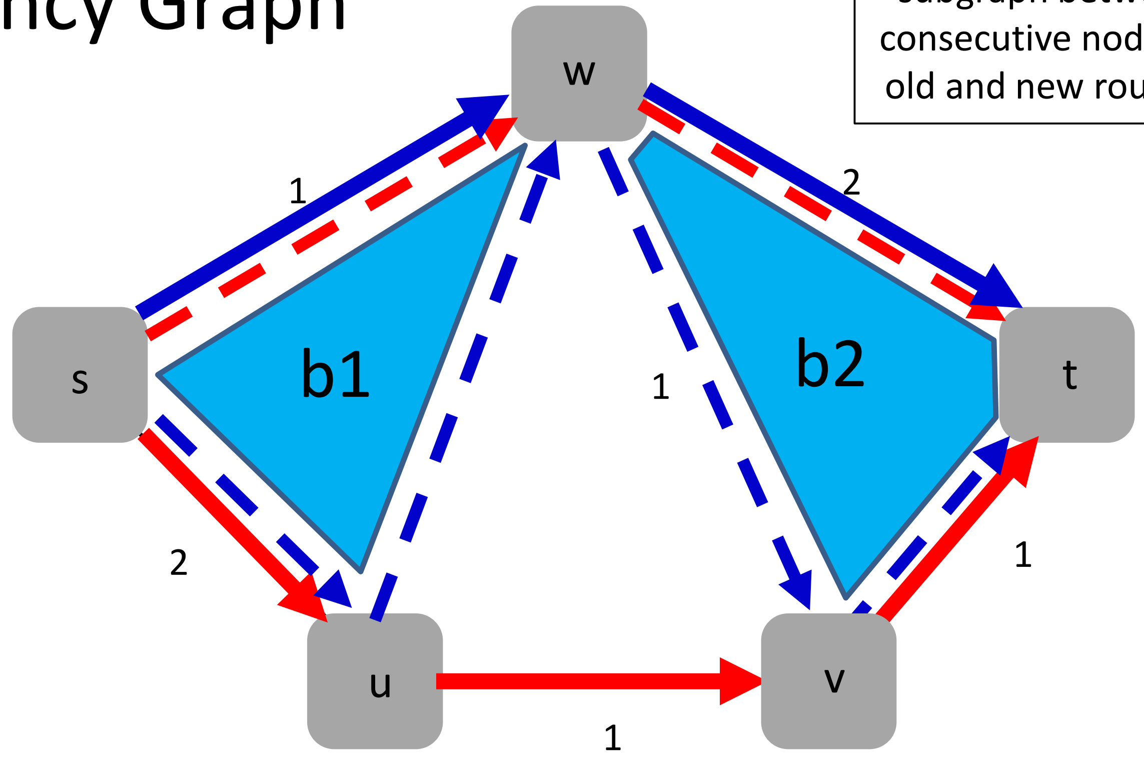
Block for a given flow: subgraph between two consecutive nodes where old and new route meet.



Just one red block: r1

Block Decomposition and Dependency Graph

Block for a given flow: subgraph between two consecutive nodes where old and new route meet.



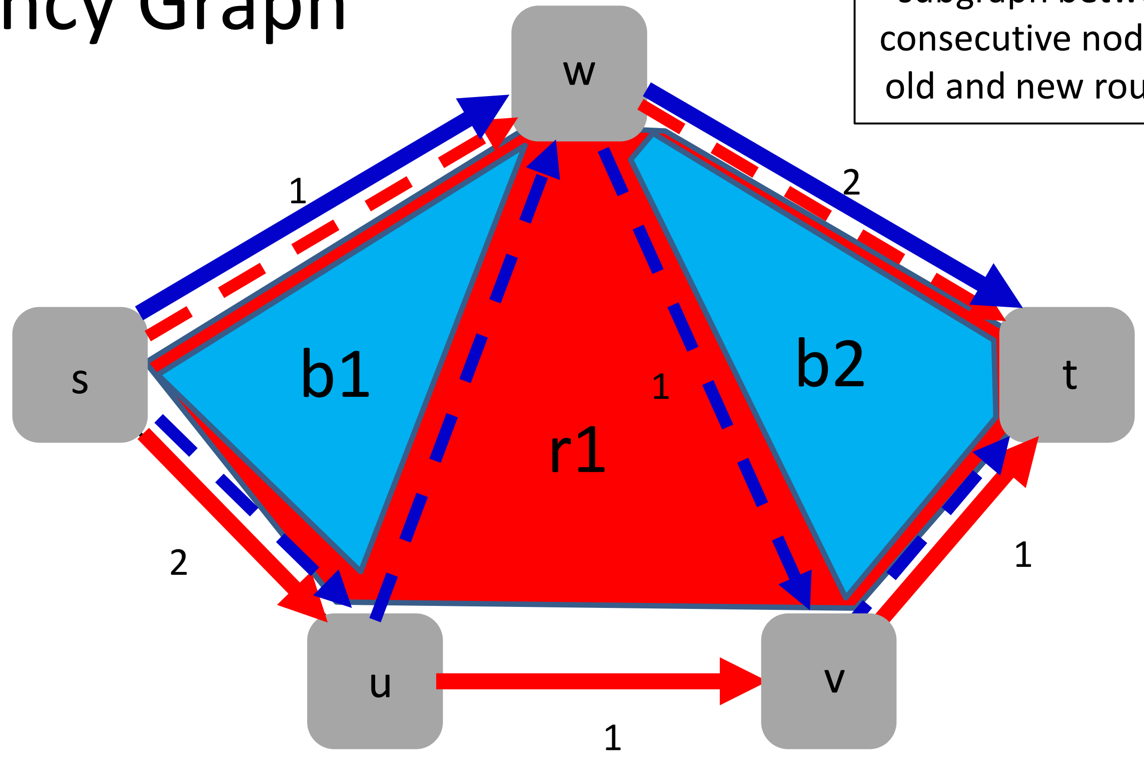
Flow 1

Flow 2

Two blue blocks: b1 and b2

Block Decomposition and Dependency Graph

Block for a given flow: subgraph between two consecutive nodes where old and new route meet.



Flow 1

Flow 2

Dependencies: update b2 after r1 after b1.

Many Open Problems

- Instance of **combinatorial reconfiguration theory** (known from ***games***)
- We know for flow graphs forming ***a DAG***:
  - For ***k=2 flows***, polynomial-time algorithm to compute schedule with **minimal number of rounds**! For general k, NP-hard.
  - For ***general constant k*** flows, polynomial-time algorithm to compute **feasible update**
- Some results for other ***transient properties*** besides congestion-freedom:
  - Transient **loop-freedom**
  - **Waypoint enforcement**
- Everything else: ***unkown!***
  - In particular: what if flow graph is not a DAG?

Further reading:
ACM PODC 2015
ACM SIGMETRICS 2016
ICALP 2018
Etc.

Many Open Problems

- Instance of **combinatorial reconfiguration theory** (known from ***games***)
- We know for flow graphs forming ***a DAG***:
  - For ***k=2 flows***, polynomial-time algorithm to compute schedule with **minimal number of rounds**! For general k, NP-hard.
  - For ***general constant k*** flows, polynomial-time algorithm to compute **feasible update**
- Some results for other ***transient properties*** besides congestion-freedom:
  - Transient **loop-freedom**
  - **Waypoint enforcement**
- Everything else: ***unkown!***
  - In particular: what if flow graph is not a DAG?

Further reading:
ACM PODC 2015
ACM SIGMETRICS 2016
ICALP 2018
Etc.

Exists research on consistency checking middleware…

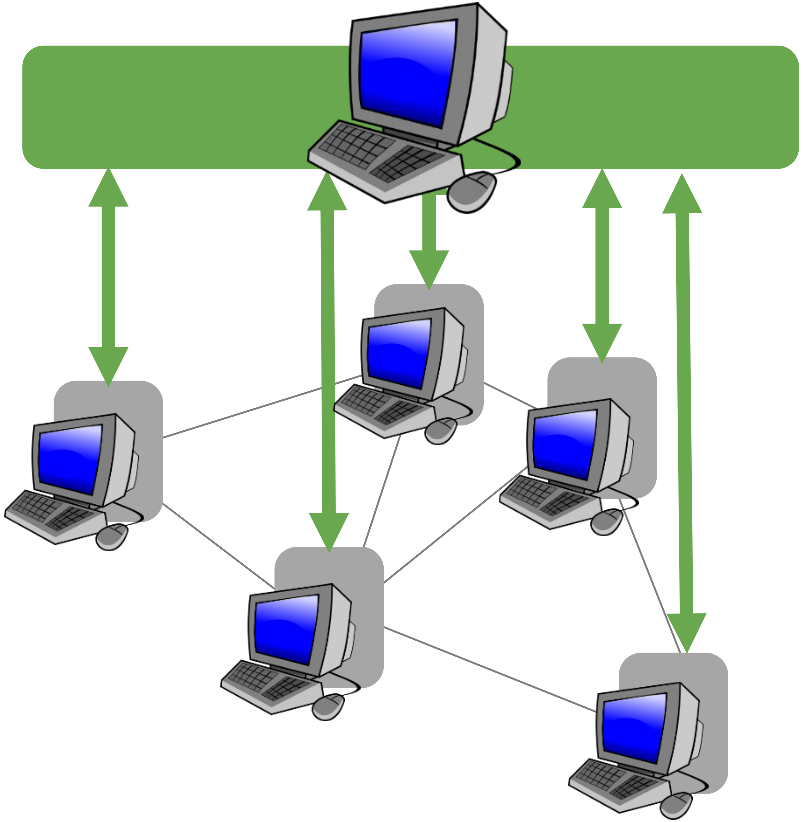
Trend: Virtualization

- Routers, switches, **middleboxes** run on commodity x86 hardware

- A.k.a. **virtual switches**

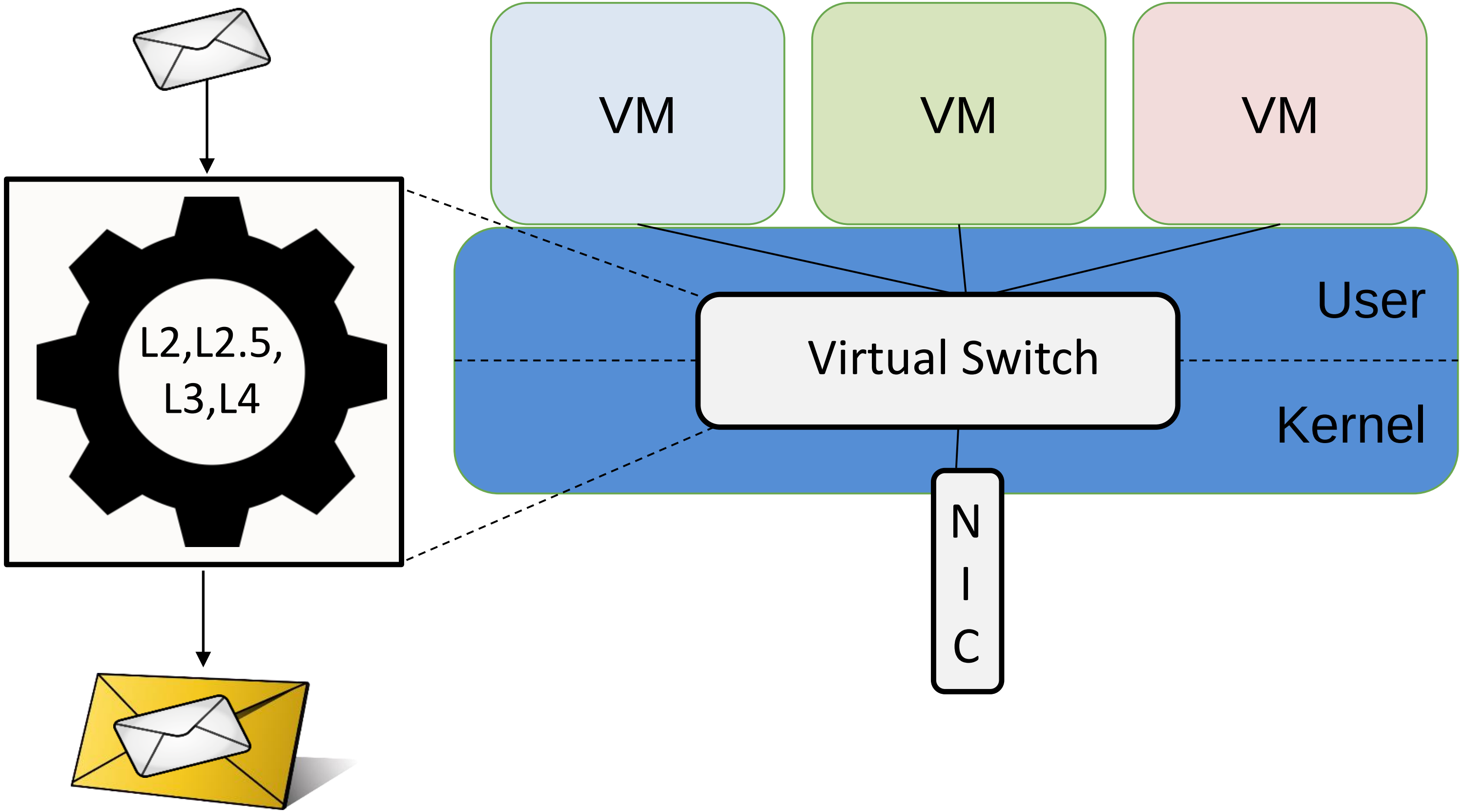
- Mainly in datacenters

- Many **complex algorithms** in the dataplane (e.g., **parsing, flow caching**): ***Uncharted security landscape!***



Virtual Switches are Complex, e.g.: (Unified) Packet Parsing

Ethernet
LLC
VLAN
MPLS
IPv4
ICMPv4
TCP
UDP
ARP
SCTP
IPv6
ICMPv6
IPv6 ND
GRE
LISP
VXLAN
PBB
IPv6 EXT HDR
TUNNEL-ID
IPv6 ND
IPv6 EXT HDR
IPv6HOPOPTS
IPv6ROUTING
IPv6Fragment
IPv6DESTOPT
IPv6ESP
IPv6 AH
RARP
IGMP

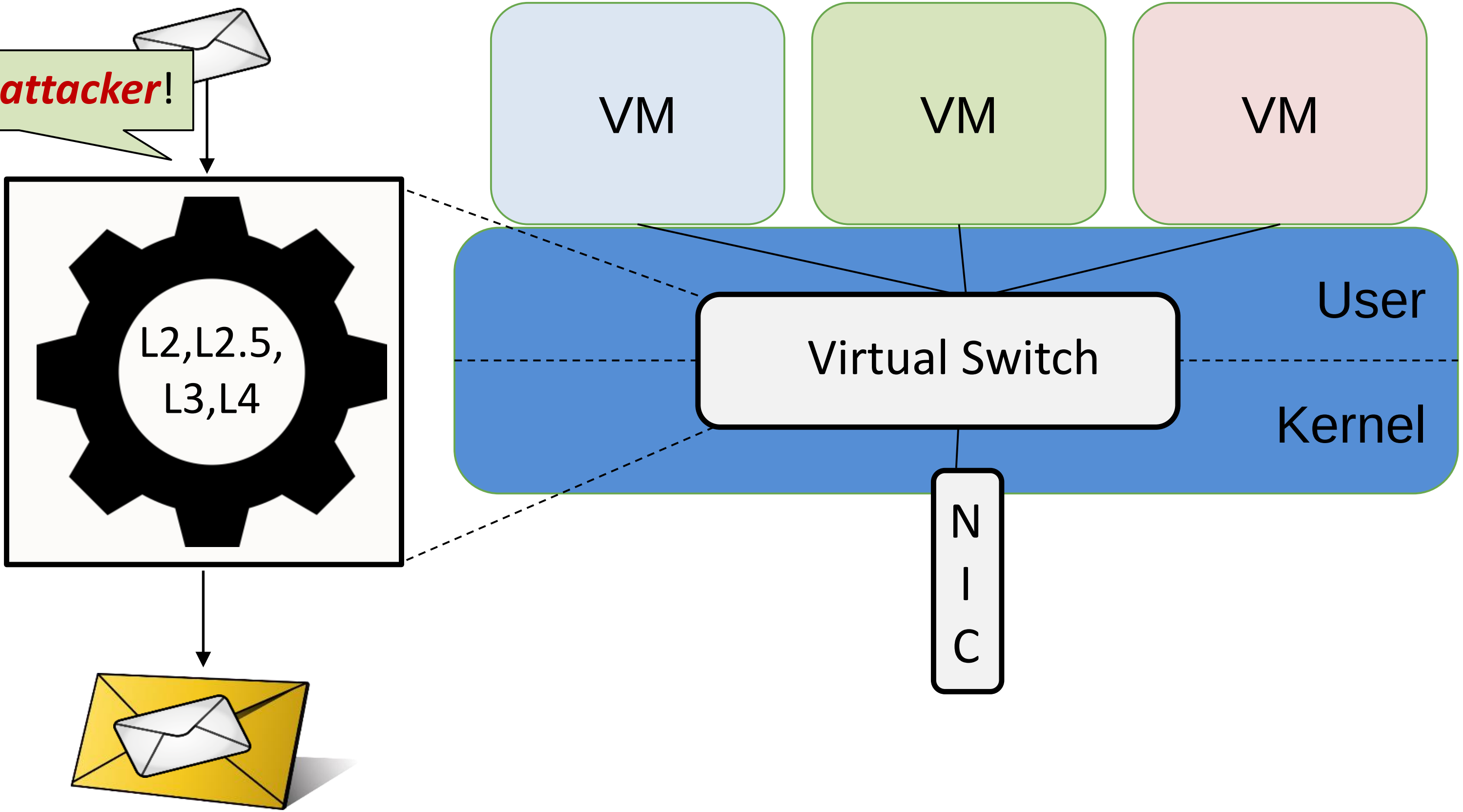


Virtual Switches are Complex, e.g.: (Unified) Packet Parsing

Ethernet
LLC
VLAN
MPLS

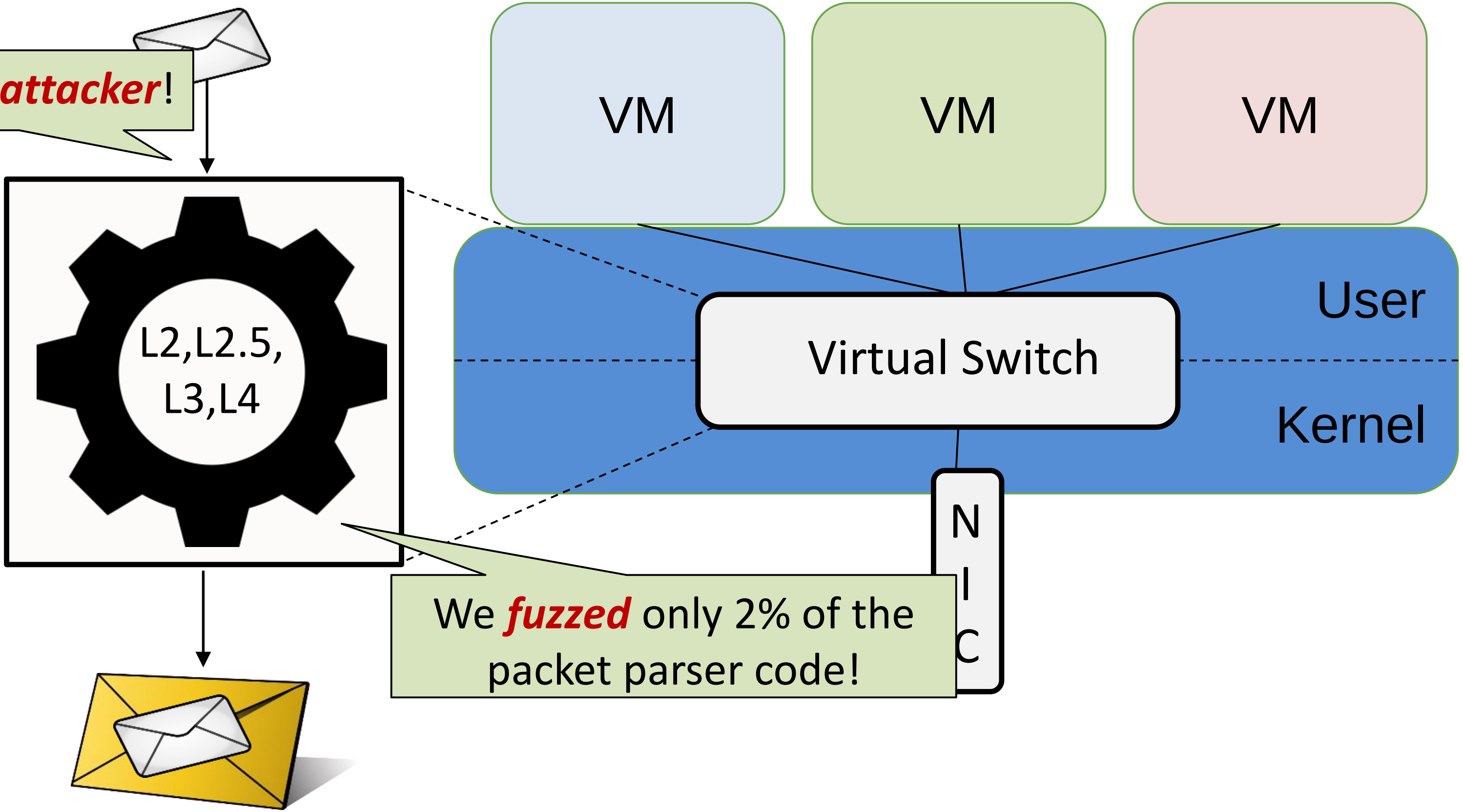
Facing the ***attacker***!

UDP
ARP
SCTP
IPv6
ICMPv6
IPv6 ND
GRE
LISP
VXLAN
PBB
IPv6 EXT HDR
TUNNEL-ID
IPv6 ND
IPv6 EXT HDR
IPv6HOPOPTS
IPv6ROUTING
IPv6Fragment
IPv6DESTOPT
IPv6ESP
IPv6 AH
RARP
IGMP



Virtual Switches are Complex, e.g.: (Unified) Packet Parsing

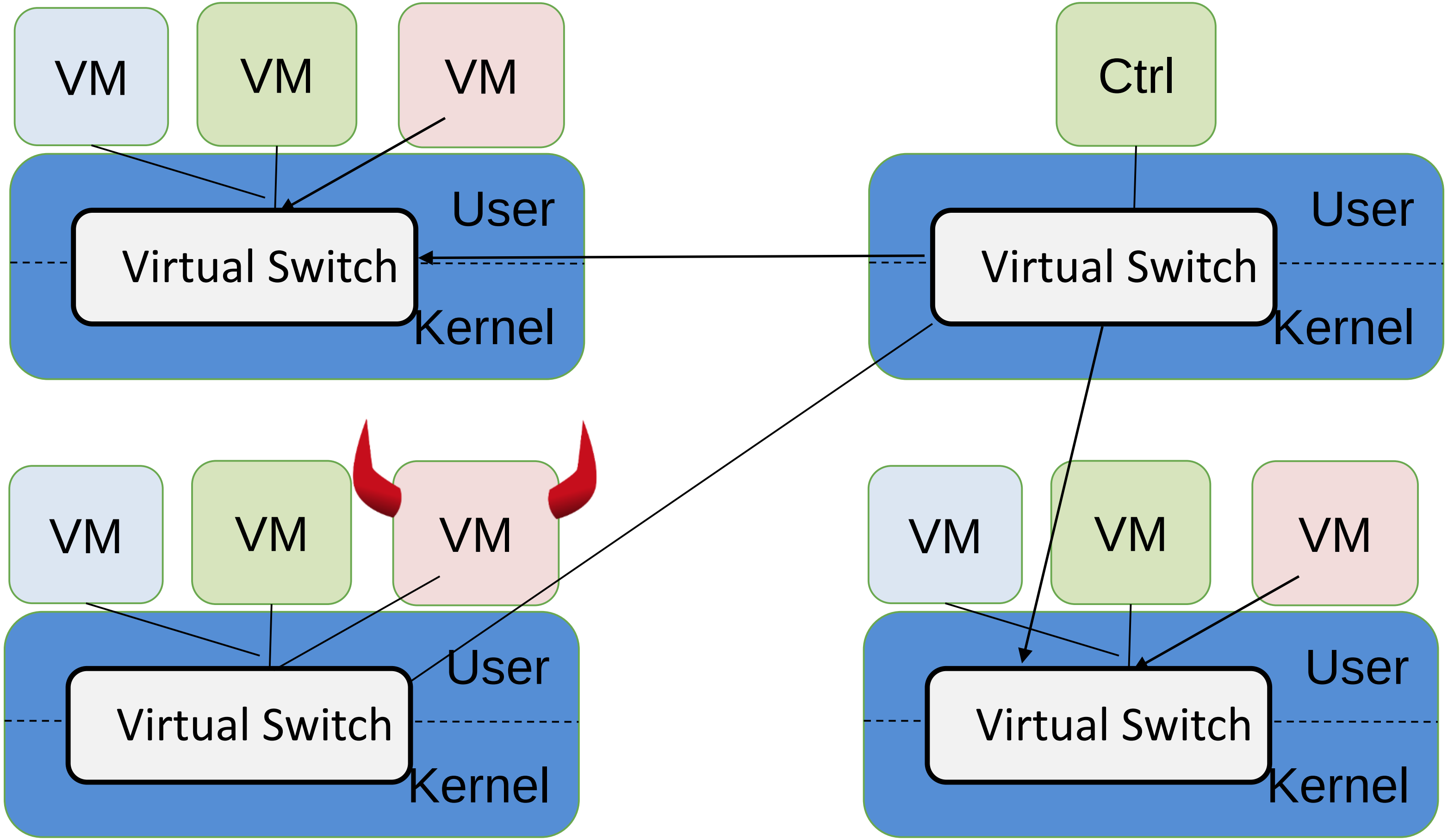
Ethernet
LLC
VLAN
MPLS
UDP
ARP
SCTP
IPv6
ICMPv6
IPv6 ND
GRE
LISP
VXLAN
PBB
IPv6 EXT HDR
TUNNEL-ID
IPv6 ND
IPv6 EXT HDR
IPv6HOPOPTS
IPv6ROUTING
IPv6Fragment
IPv6DESTOPT
IPv6ESP
IPv6 AH
RARP
IGMP



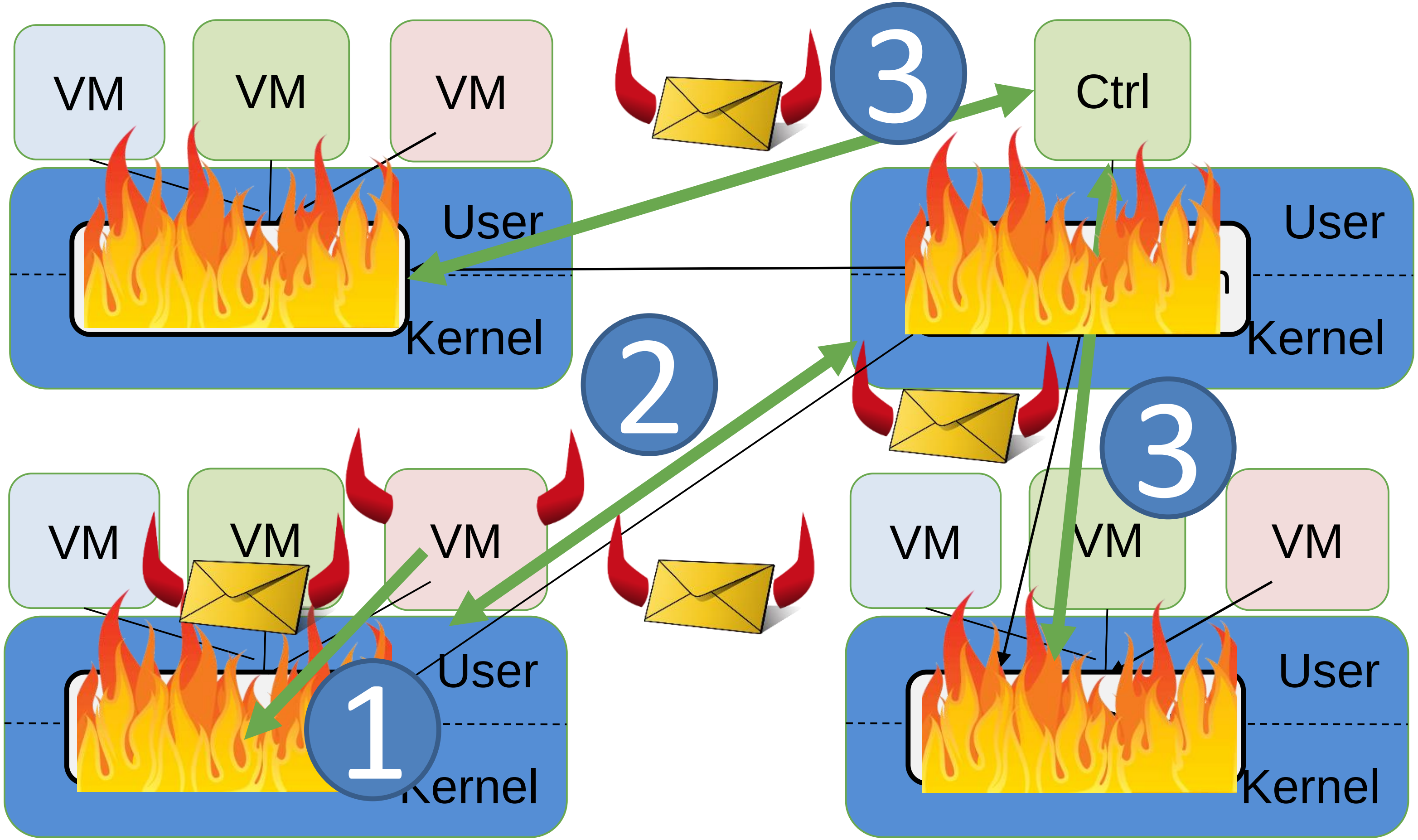
Facing the ***attacker***!

We ***fuzzed*** only 2% of the packet parser code!

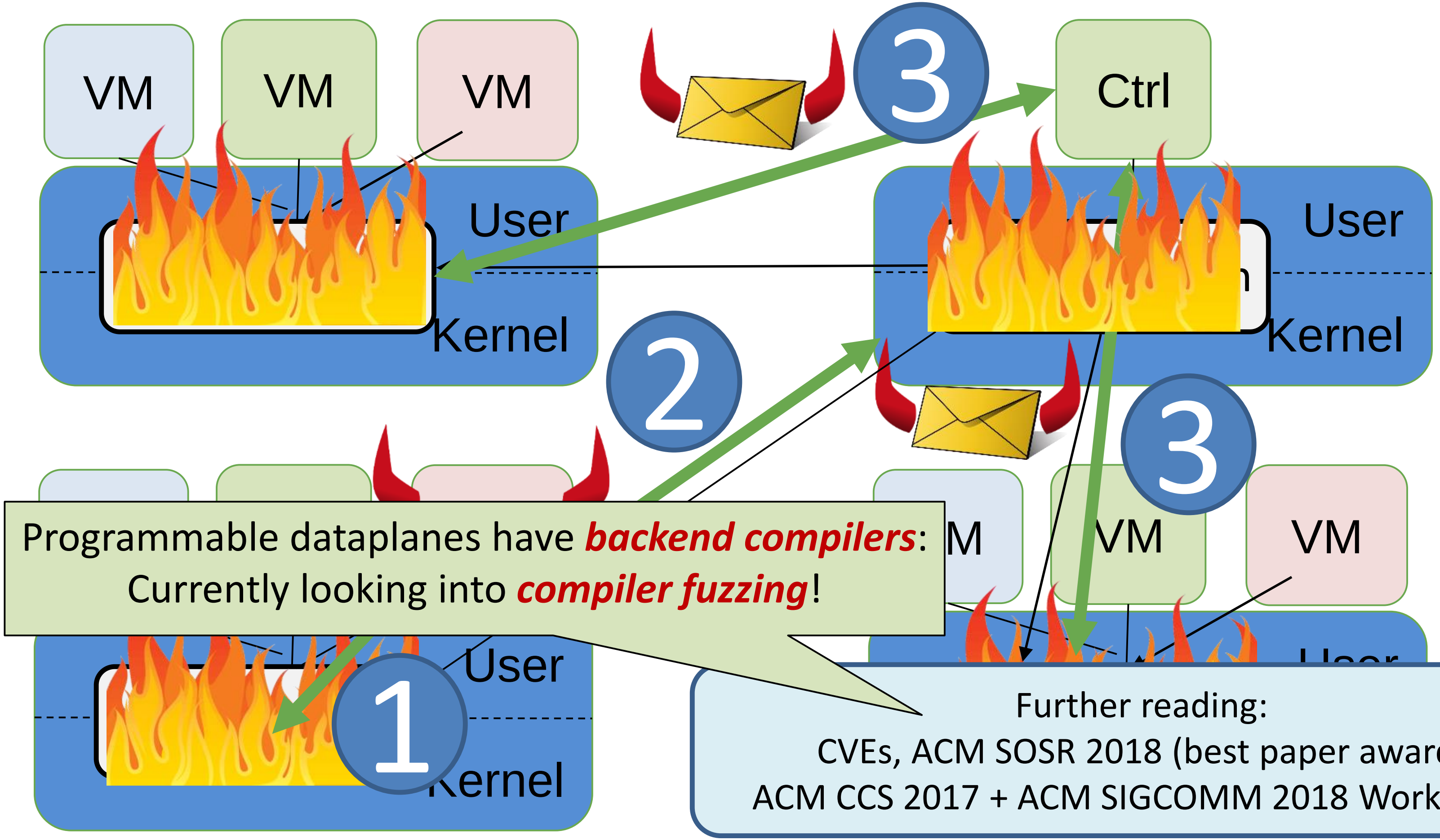
Compromising the Cloud



Compromising the Cloud



Compromising the Cloud



Programmable dataplanes have ***backend compilers***:
Currently looking into ***compiler fuzzing***!

Further reading:
CVEs, ACM SOSR 2018 (best paper award)
ACM CCS 2017 + ACM SIGCOMM 2018 Workshops

Automated Network Verification

- Recent years: growing interest in high-level **languages for programming networks**, some ***ad-hoc***…

- … some with solid ***semantic foundations***

- E.g., **NetKAT**: sound and complete equational theory
  - Primitives for ***filtering, modifying and transmitting packets***
  - An instance of **Kleene algebra with Tests** (KAT)
  - Can be used, e.g., for checking **reachability**

Automated Network Verification

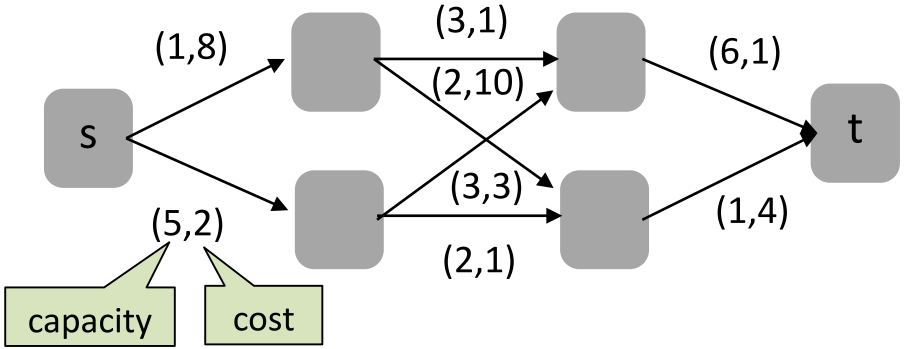
- Recent years: growing interest in high-level **languages for programming networks**, some ***ad-hoc***…
- … some with solid ***semantic foundations***
- E.g., **NetKAT**: sound and comple
  - Primitives for ***filtering, modifying an***
  - An instance of **Kleene algebra with T**
  - Can be used, e.g., for checking **rea** **y**

Perhaps the hottest topic in networking these days…: Nate and Dexter's papers highly cited.

WNetKAT

WNetKAT. Kim G. Larsen, Stefan Schmid, and Bingtian Xue. OPODIS 2016.

- A **weighted** SDN programming and verification language
- Goes ***beyond topological*** aspects but account for:
  - actual **resource** availabilities, **capacities**, **costs**, or even **stateful** operations

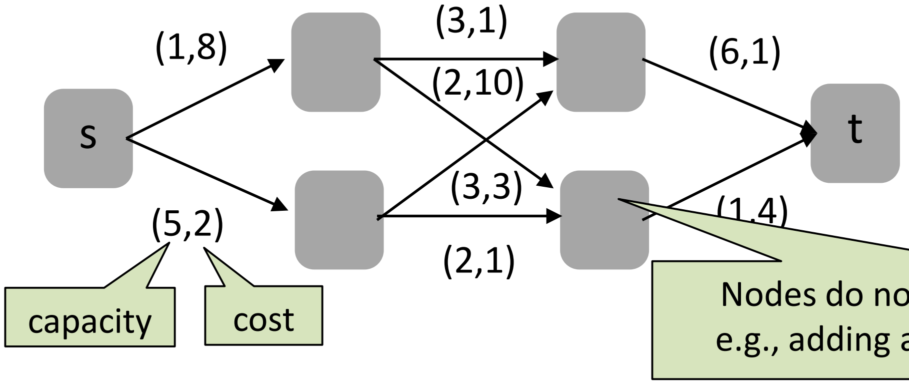


E.g.: Can s reach t at cost/bandwidth/latency x?

WNetKAT

WNetKAT. Kim G. Larsen, Stefan Schmid, and Bingtian Xue. OPODIS 2016.

- A **weighted** SDN programming and verification language
- Goes ***beyond topological*** aspects but account for:
  - actual **resource** availabilities, **capacities**, **costs**, or even **stateful** operations



E.g.: Can s reach t at cost/bandwidth/latency x?

Nodes do not have to be **flow-conserving**: e.g., adding a packet header for ***tunneling***!

The Good News

- Networks are becoming more **programmable** and logically **centralized**, have **open** interfaces, …
- … are based on **formal foundations**
- Enables a more **automated** network operation and verification!

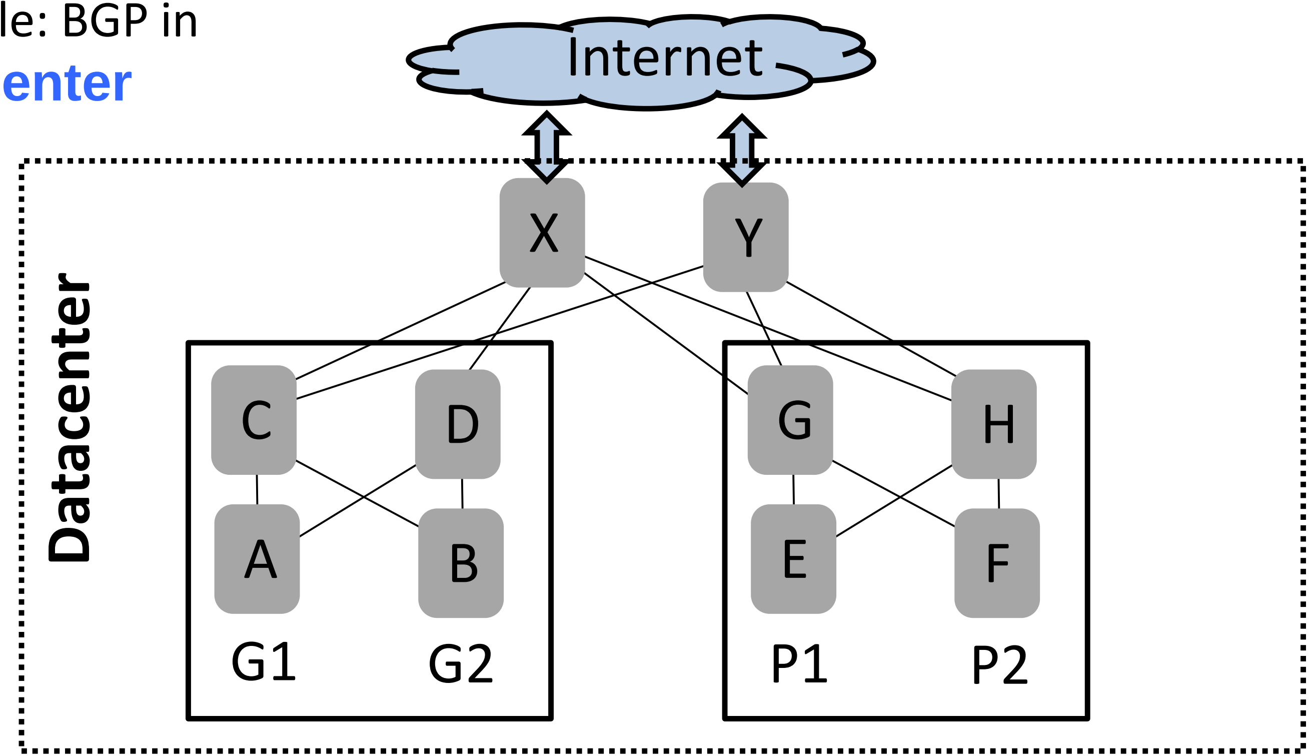
The Bad News

- For many **traditional networks** (still *predominant*!), such benefits are not available yet
- **Super-polynomial time** for verification: ***PSPACE-hard*** (NetKAT) or even ***undecidable*** (WNetKAT)
- Other limitations: e.g., **fixed header size**

Things get more complex when one wants to check properties ***under failures***.

A Challenge: Reachability Under Failures

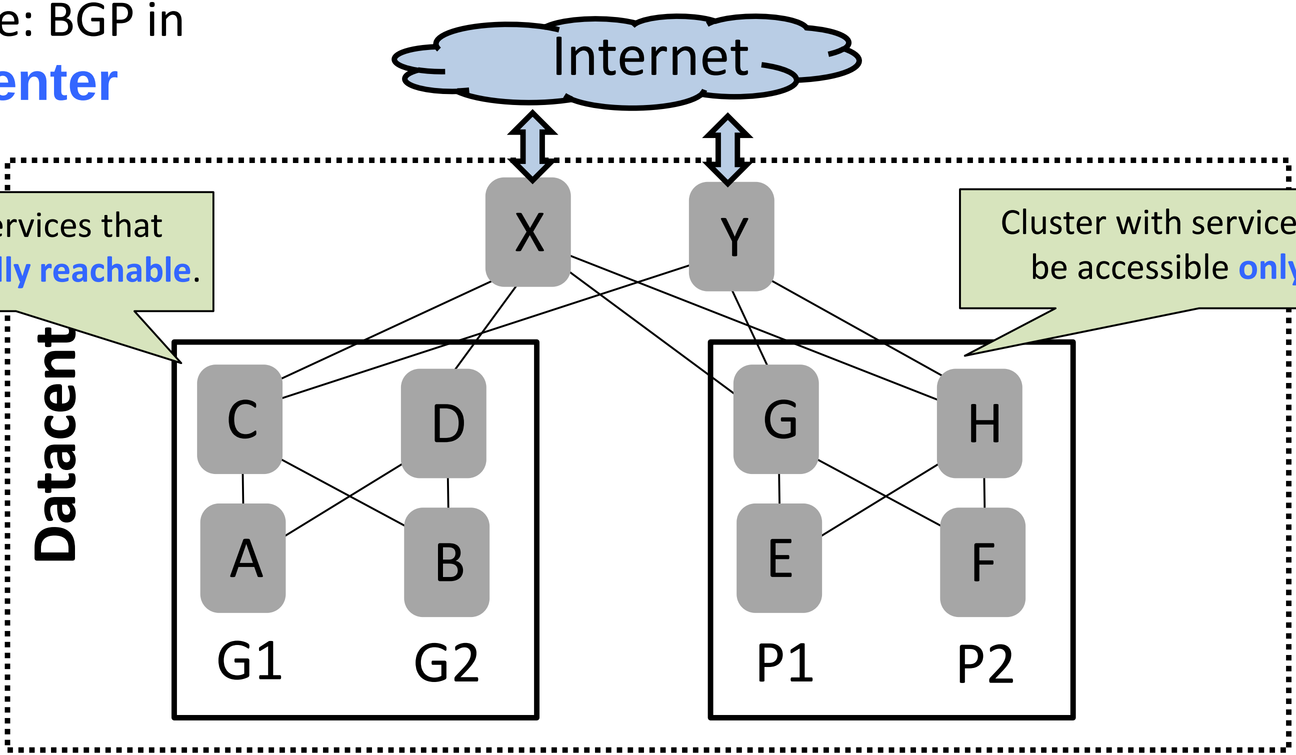
Example: BGP in
Datacenter



Credits: Beckett et al. (SIGCOMM 2016): Bridging Network-wide Objectives and Device-level Configurations.

A Challenge: Reachability Under Failures

Example: BGP in **Datacenter**

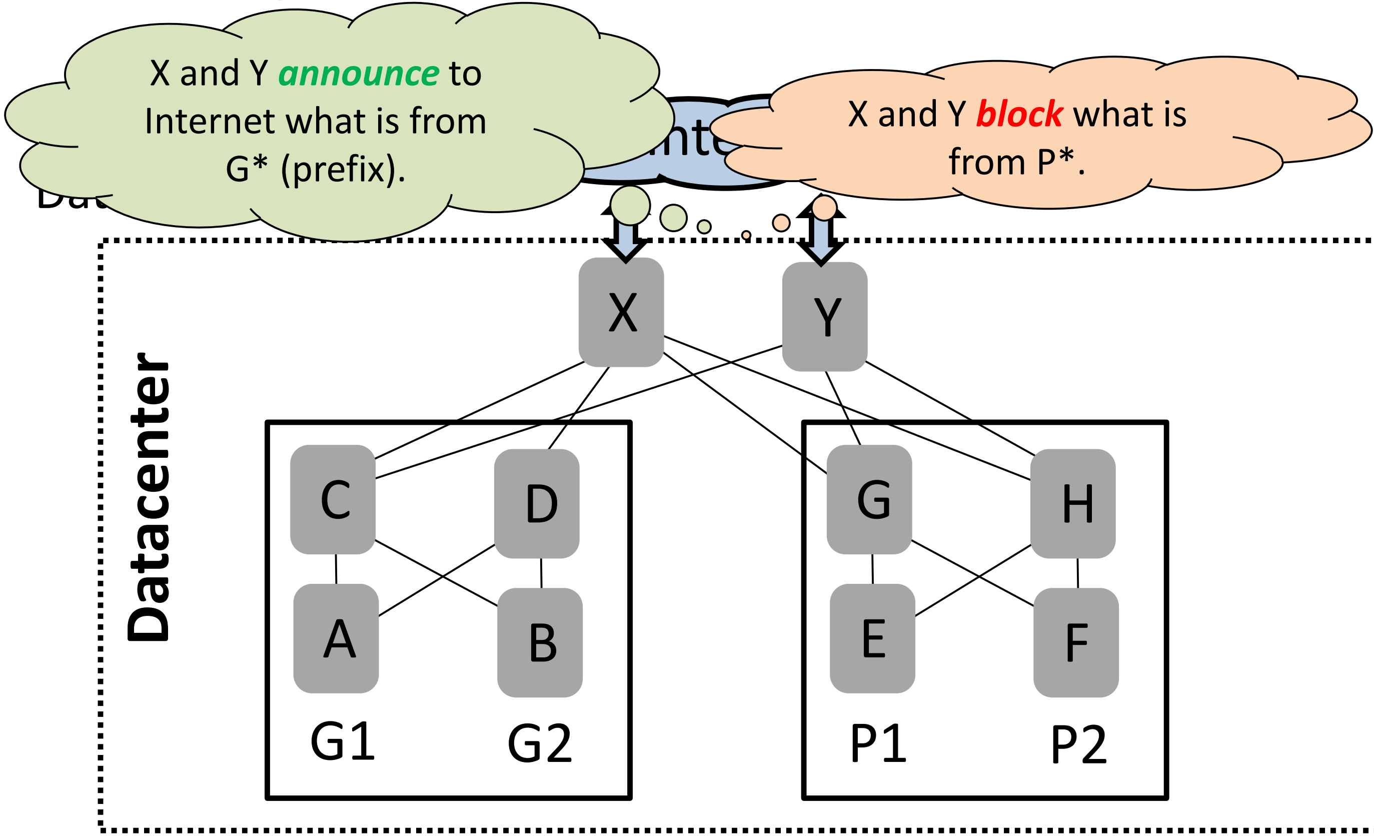


Cluster with services that should be **globally reachable**.

Cluster with services that should be accessible **only internally**.

Credits: Beckett et al. (SIGCOMM 2016): Bridging Network-wide Objectives and Device-level Configurations.

A Challenge: Reachability Under Failures

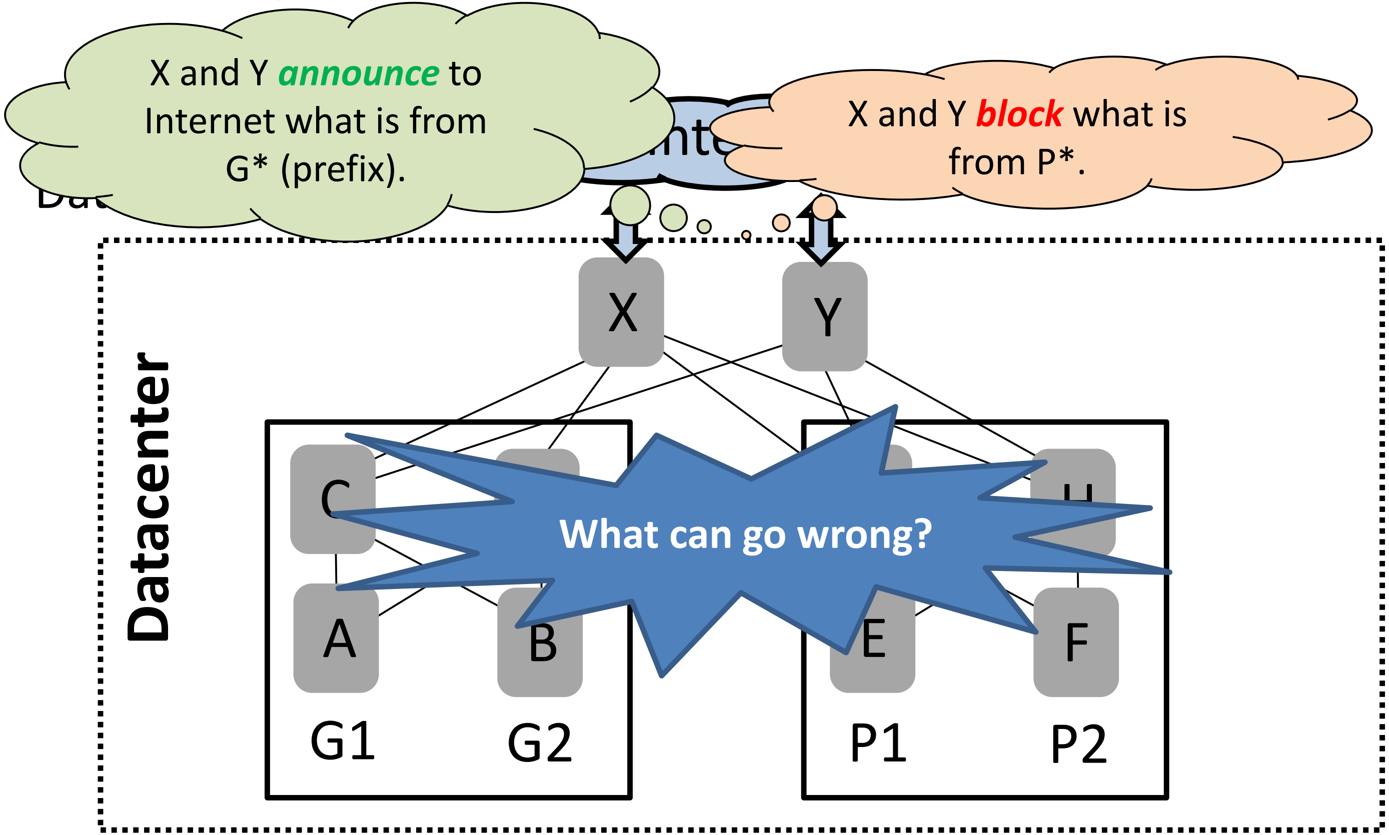

X and Y *announce* to Internet what is from G* (prefix).
X and Y *block* what is from P*.
Inte
Dat
Datacenter
X
Y
C
D
A
B
G
H
E
F
G1
G2
P1
P2

Credits: Beckett et al. (SIGCOMM 2016): Bridging Network-wide Objectives and Device-level Configurations.

A Challenge: Reachability Under Failures

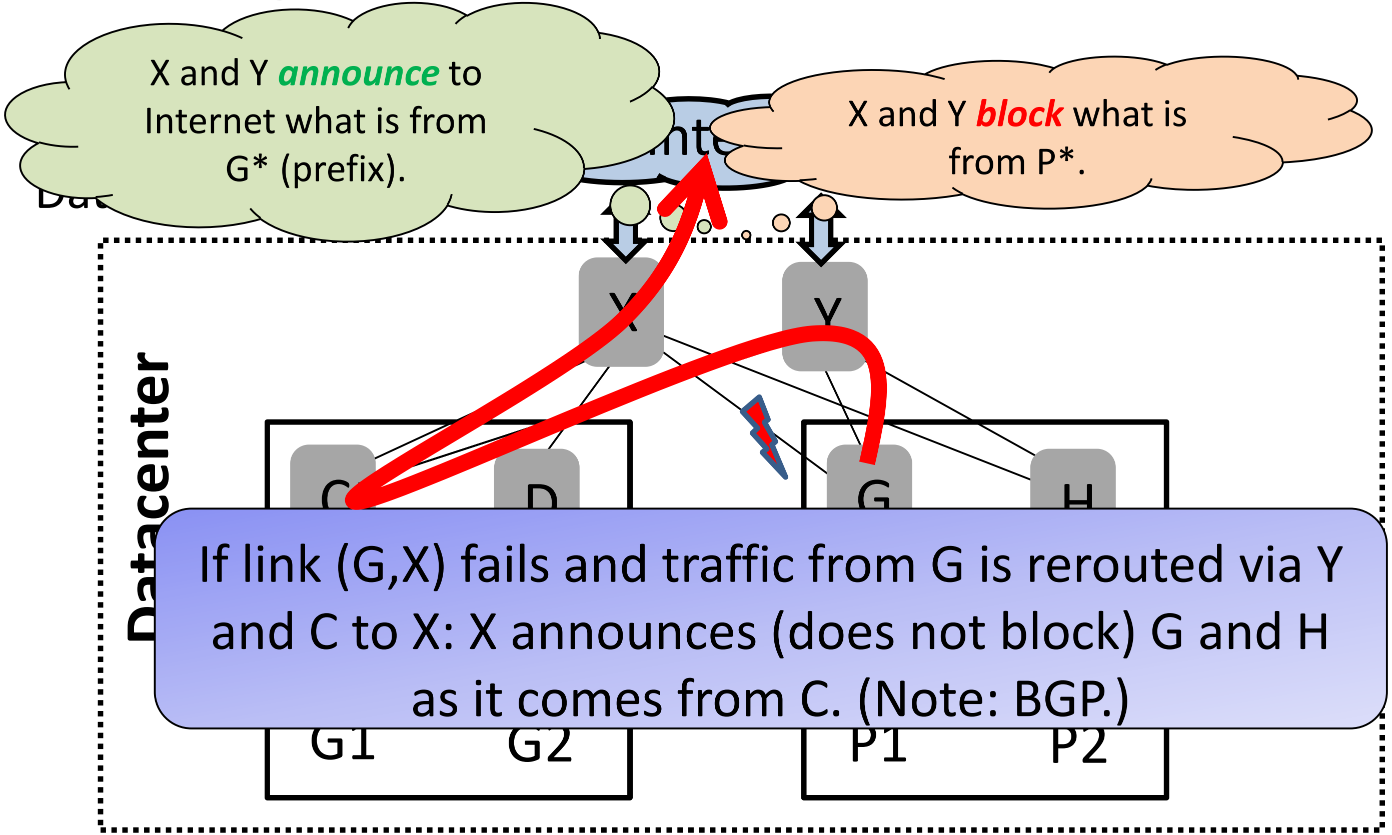
X and Y ***announce*** to Internet what is from G* (prefix).

X and Y ***block*** what is from P*.



Credits: Beckett et al. (SIGCOMM 2016): Bridging Network-wide Objectives and Device-level Configurations.

A Challenge: Reachability Under Failures



X and Y *announce* to Internet what is from G* (prefix).
X and Y *block* what is from P*.
X
Y
C
D
G
H
G1
G2
P1
P2
Datacenter

If link (G,X) fails and traffic from G is rerouted via Y and C to X: X announces (does not block) G and H as it comes from C. (Note: BGP.)

Credits: Beckett et al. (SIGCOMM 2016): Bridging Network-wide Objectives and Device-level Configurations.

Our Contribution

Independently of the number of failures! No need to try combinations.

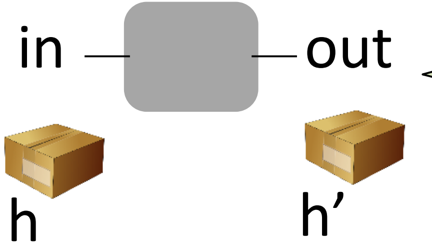
Reachability, loop-freedom, waypointing, etc.!

Polynomial-Time What-if Analysis for Prefix Rewriting Networks

Case Study: MPLS networks or Segment Routing networks. **Widely deployed** by ISPs!

MPLS and SR: Special Rules

The **clue**: exploit the specific structure of MPLS rules.



Rules match the header **h** of packets arriving at ***in***, and define to which port ***out*** to forward as well as new header **h'**.

Rules of general networks (e.g., SDN):

arbitrary header rewriting

$in \times L^* \rightarrow out \times L^*$

VS

(Simplified) MPLS rules:

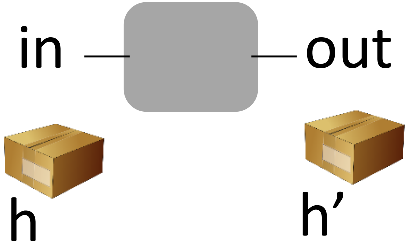
prefix rewriting

$in \times L \rightarrow out \times \mathbf{OP}$

where *OP* = {*swap,push,pop*}

MPLS and SR: Special Rules

The **clue**: exploit the specific structure of MPLS rules.



Rules match the header **h** of packets arriving at ***in***, and define to which port ***out*** to forward as well as new header **h’**.

Rules of general netwo g., SDN):

arbitrary hea ewriting

*in x L → out x L**

Undecidable!

vs

(Simplified) MP s:

prefix r ing

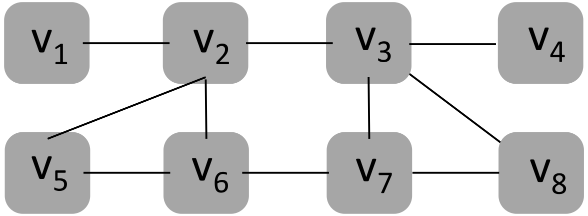
→ out x ***OP***

wh OP = {swap,push,pop}

Polynomial-time!

How MPLS Networks Work

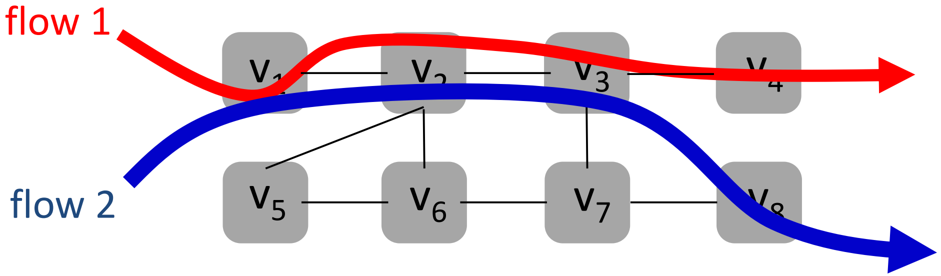
- MPLS: forwarding based on **top label** of label **stack**



Default routing of two flows

How MPLS Networks Work

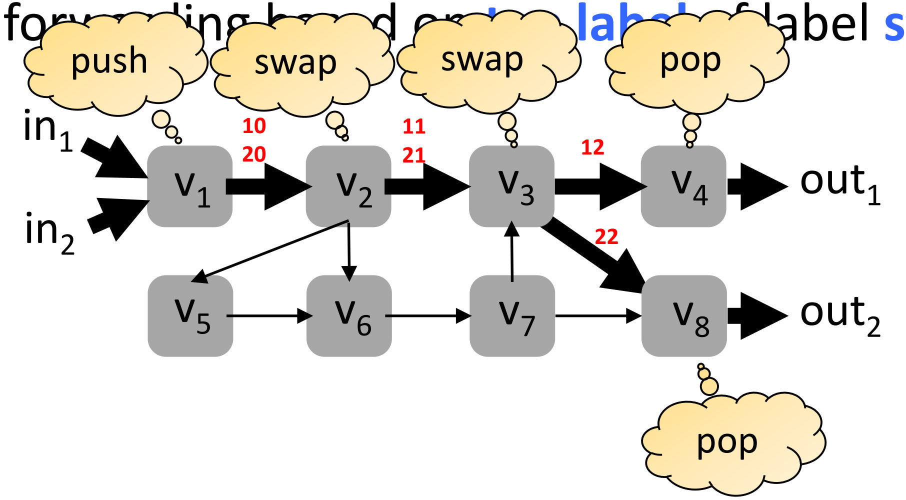
- MPLS: forwarding based on **top label** of label **stack**



Default routing of
two flows

How MPLS Networks Work

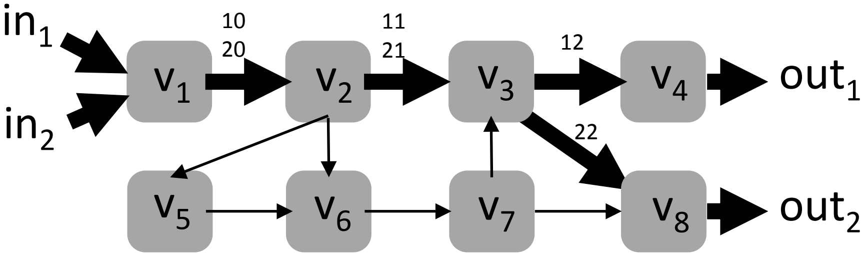
- MPLS: forwarding based on **top label** of label **stack**



Default routing of two flows

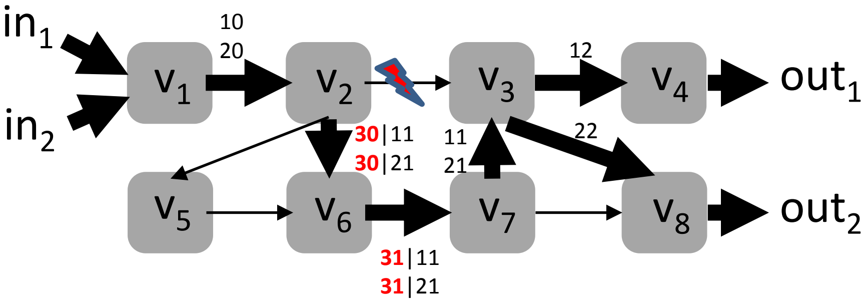
Fast Reroute Around *1 Failure*

- MPLS: forwarding based on **top label** of label **stack**



Default routing of two flows

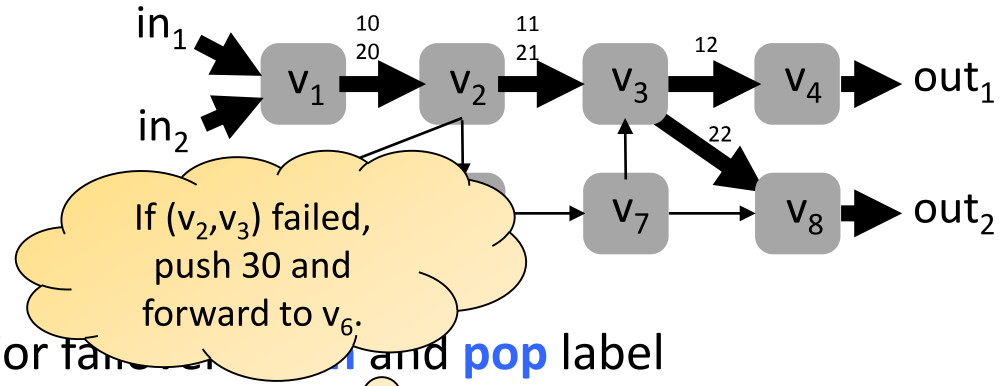
- For failover: **push** and **pop** label



One failure: push 30: route around (v_2,v_3)

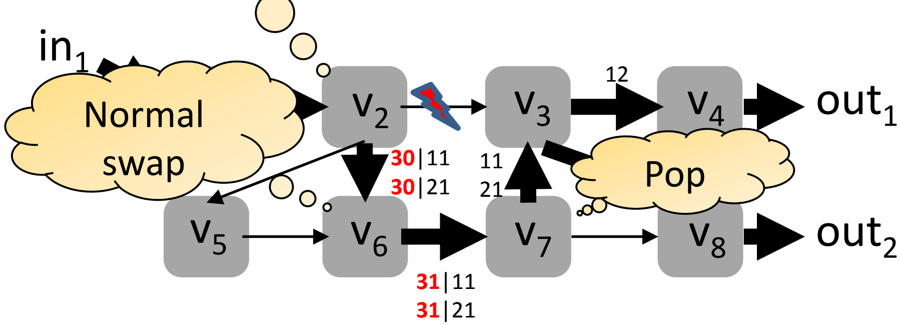
Fast Reroute Around *1 Failure*

- MPLS: forwarding based on **top label** of label **stack**



Default routing of two flows

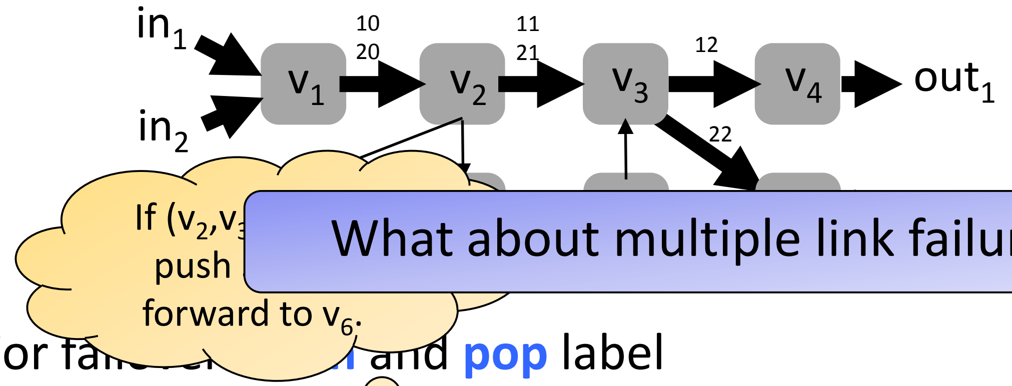
- For fa[illegible] and **pop** label



One failure: push 30: route around (v$_2$,v$_3$)

Fast Reroute Around *1 Failure*

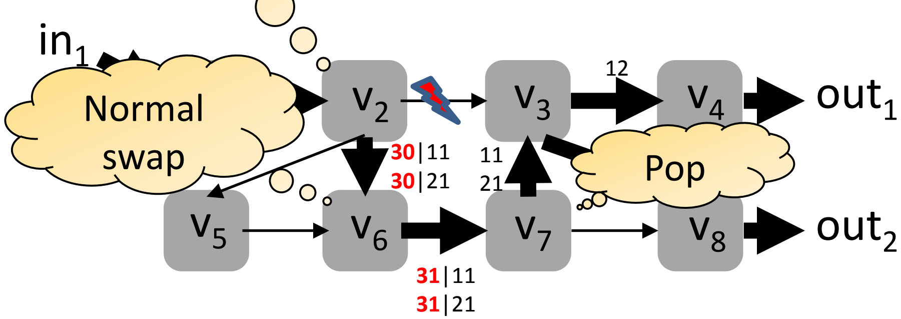
- MPLS: forwarding based on **top label** of label **stack**



Default routing of two flows

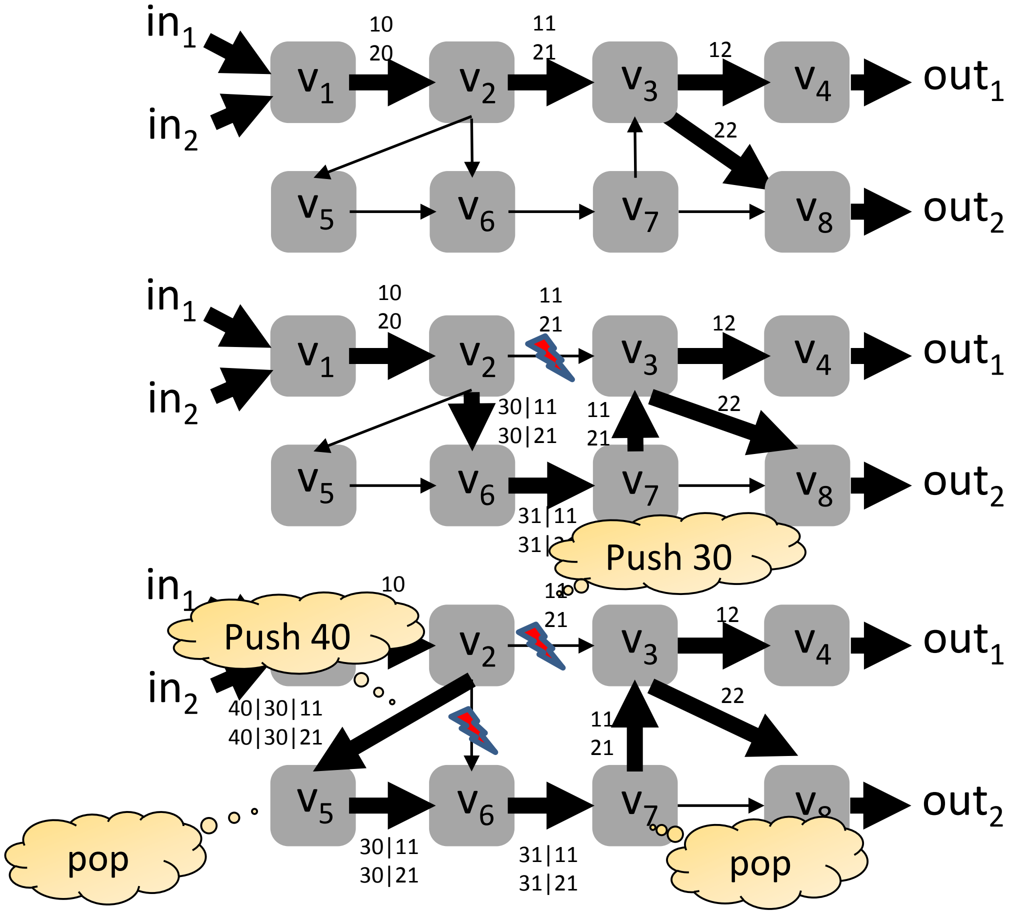
What about multiple link failures?

- For fa and **pop** label



One failure: push 30: route around (v_2,v_3)

2 Failures: Push *Recursively*



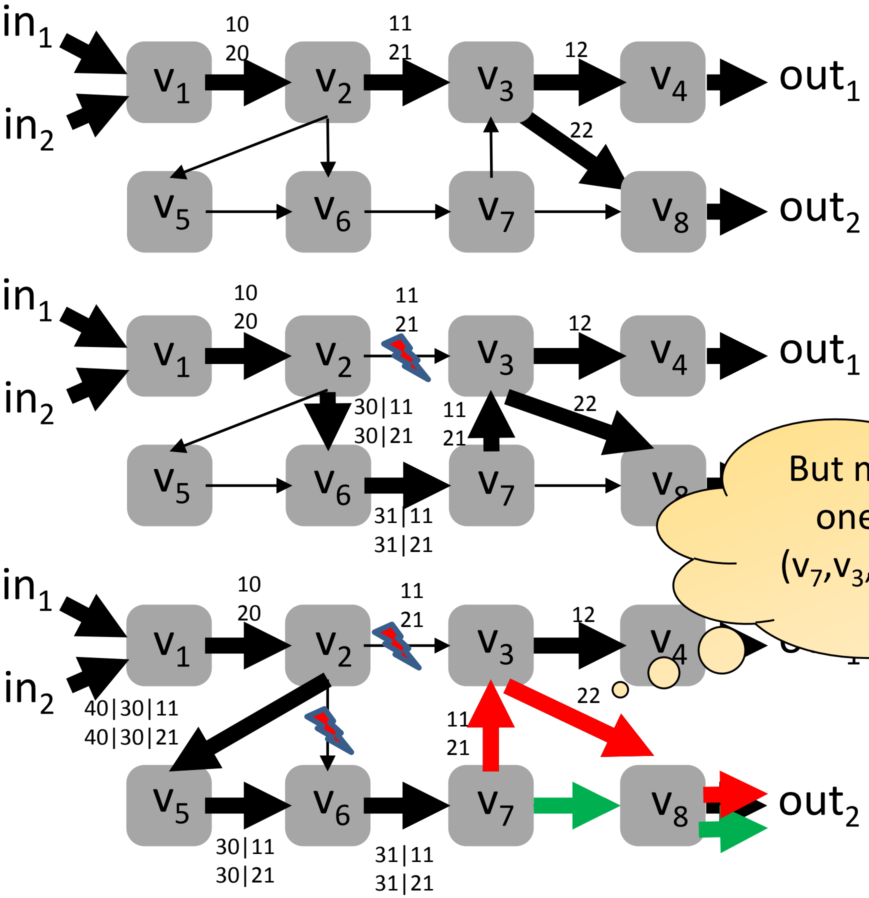
Original Routing

One failure: push 30: route around (v_2,v_3)

Two failures:
first push 30: route around (v_2,v_3)

Push recursively 40: route around (v_2,v_6)

2 Failures: Push *Recursively*



Original Routing

One failure: push 30:

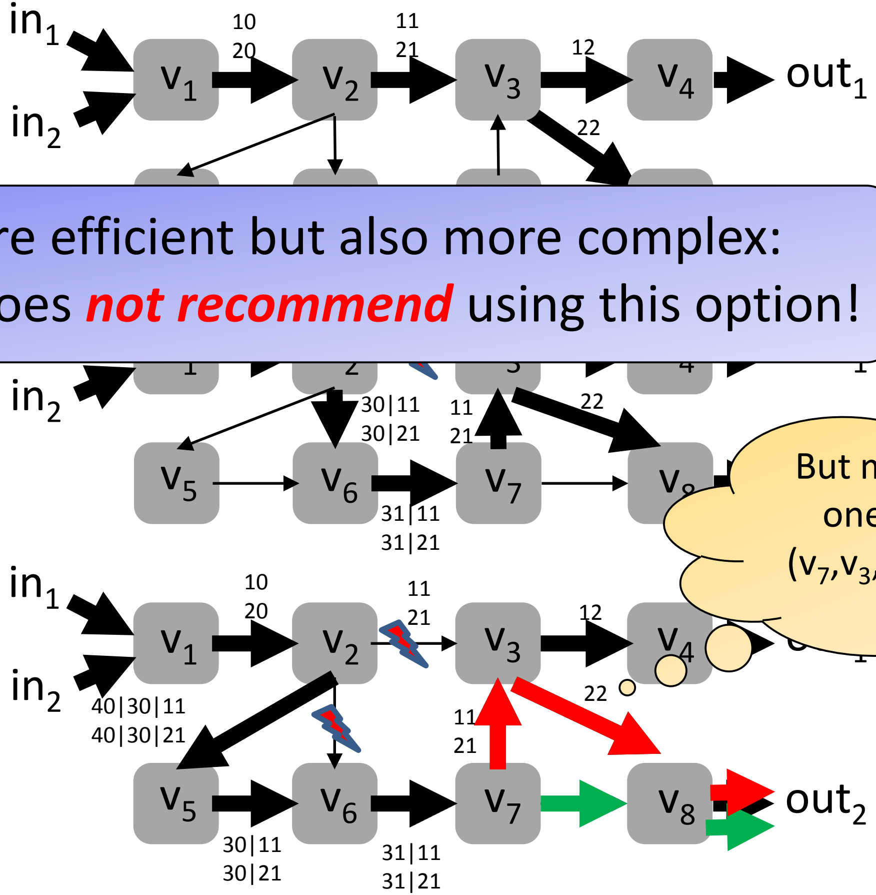
)

But masking links one-by-one can be inefficient: (v_7,v_3,v_8) could be shortcut to (v_7,v_8).

sh 30: route around (v_2,v_3)

Push recursively 40: route around (v_2,v_6)

2 Failures: Push *Recursively*



More efficient but also more complex:
Cisco does ***not recommend*** using this option!

Original Routing

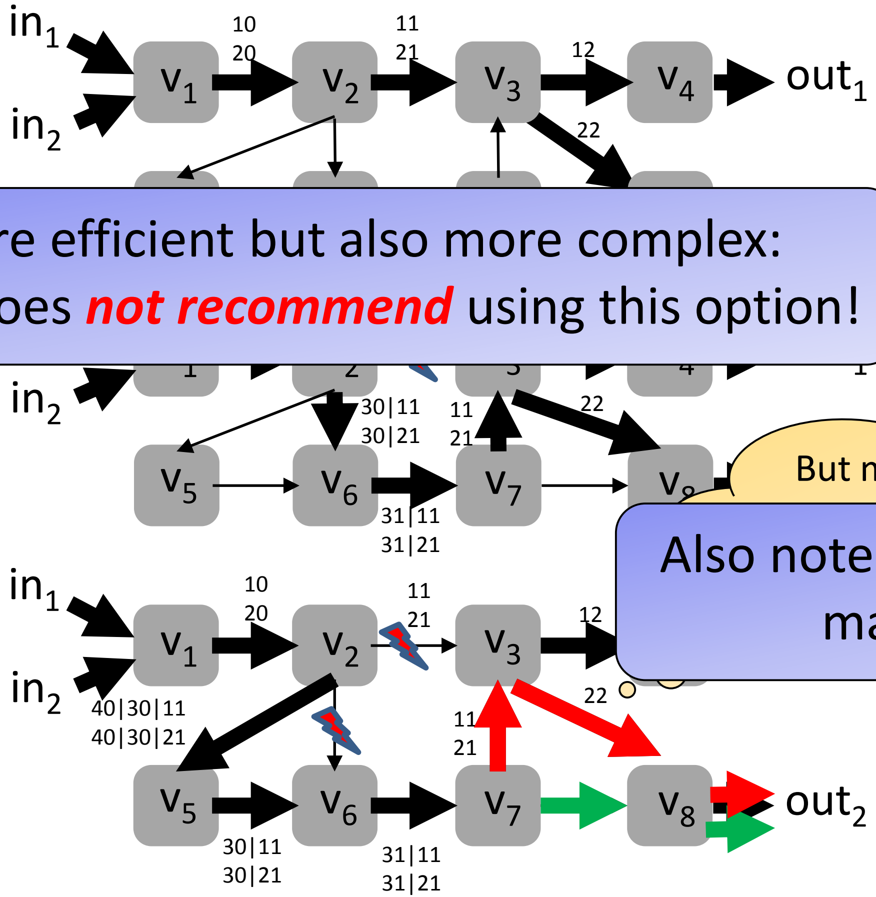
One failure: push 30:

But masking links one-by-one can be inefficient: (v_7,v_3,v_8) could be shortcut to (v_7,v_8).

sh 30: route around (v_2,v_3)

Push recursively 40: route around (v_2,v_6)

2 Failures: Push *Recursively*



Original Routing

More efficient but also more complex:
Cisco does ***not recommend*** using this option!

One failure: push 30:

But masking links one-by-

Also note: due to push, ***header size*** may grow arbitrarily!

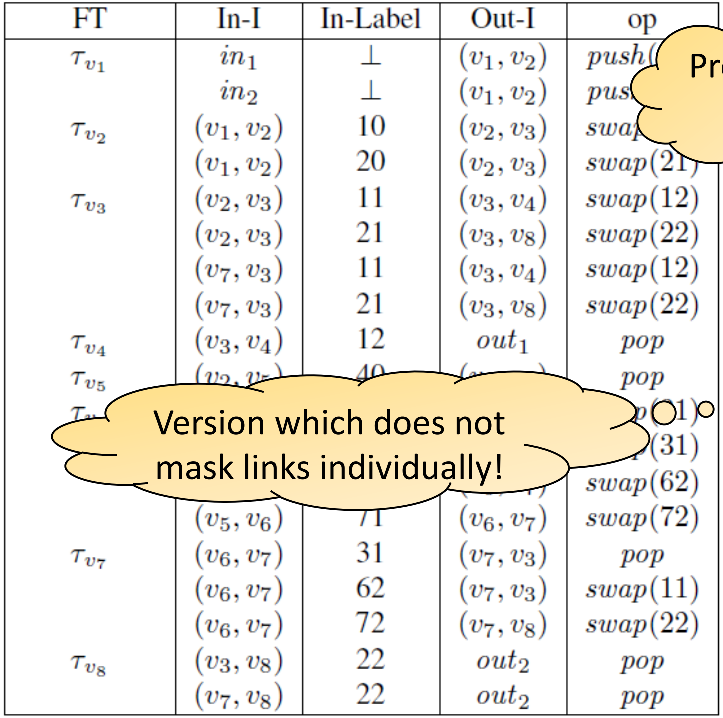
around (v_2,v_3)

Push recursively 40:
route around (v_2,v_6)

Survey: MPLS Tunnels in Today‘s ISP Networks

Survey: MPLS Tunnels in Today‘s ISP Networks

Forwarding Tables for Our Example



FT	In-I	In-Label	Out-I	op
τ_{v_1}	in_1	$\perp$	(v_1, v_2)	$push($[illegible]
	in_2	$\perp$	(v_1, v_2)	pus[illegible]
τ_{v_2}	(v_1, v_2)	10	(v_2, v_3)	swa[illegible]
	(v_1, v_2)	20	(v_2, v_3)	$swap(21)$
τ_{v_3}	(v_2, v_3)	11	(v_3, v_4)	$swap(12)$
	(v_2, v_3)	21	(v_3, v_8)	$swap(22)$
	(v_7, v_3)	11	(v_3, v_4)	$swap(12)$
	(v_7, v_3)	21	(v_3, v_8)	$swap(22)$
τ_{v_4}	(v_3, v_4)	12	out_1	pop
τ_{v_5}	(v_2, v_5)	40	[illegible]	pop
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible](31)
	[illegible]	[illegible]	[illegible]	$swap(62)$
	(v_5, v_6)	71	(v_6, v_7)	$swap(72)$
τ_{v_7}	(v_6, v_7)	31	(v_7, v_3)	pop
	(v_6, v_7)	62	(v_7, v_3)	$swap(11)$
	(v_6, v_7)	72	(v_7, v_8)	$swap(22)$
τ_{v_8}	(v_3, v_8)	22	out_2	pop
	(v_7, v_8)	22	out_2	pop

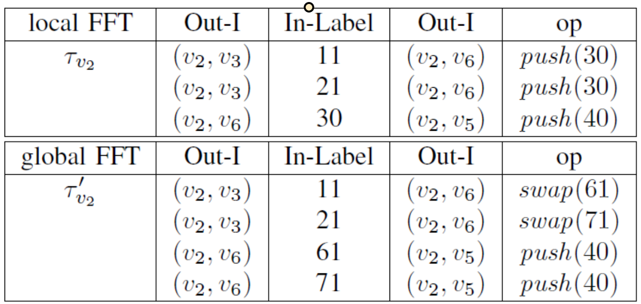
Version which does not mask links individually!

Flow Table

Protected link

Alternative link

Label



local FFT	Out-I	In-Label	Out-I	op
τ_{v_2}	(v_2, v_3)	11	(v_2, v_6)	$push(30)$
	(v_2, v_3)	21	(v_2, v_6)	$push(30)$
	(v_2, v_6)	30	(v_2, v_5)	$push(40)$

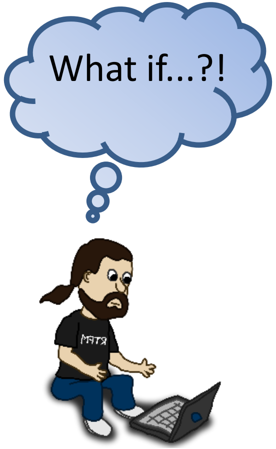
global FFT	Out-I	In-Label	Out-I	op
τ'_{v_2}	(v_2, v_3)	11	(v_2, v_6)	$swap(61)$
	(v_2, v_3)	21	(v_2, v_6)	$swap(71)$
	(v_2, v_6)	61	(v_2, v_5)	$push(40)$
	(v_2, v_6)	71	(v_2, v_5)	$push(40)$

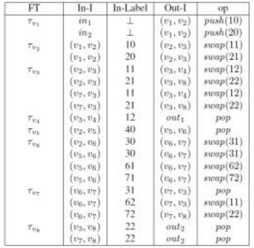
Failover Tables

The Key Insight

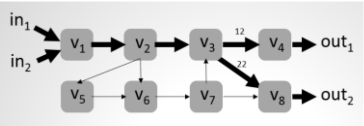
We can model MPLS networks using a context-free language (push-down automaton)! Or more specifically: A ***Prefix Rewriting System***.

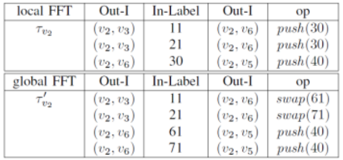
Polynomial-Time Verification: An Automata-Theoretic Approach





FT	In-I	In-Label	Out-I	op
τ_{v_1}	in_1	$\perp$	(v_1, v_2)	$push(10)$
	in_2	$\perp$	(v_1, v_2)	$push(20)$
τ_{v_2}	(v_1, v_2)	10	(v_2, v_3)	$swap(11)$
	(v_1, v_2)	20	(v_2, v_3)	$swap(21)$
τ_{v_3}	(v_2, v_3)	11	(v_3, v_4)	$swap(12)$
	(v_2, v_3)	21	(v_3, v_8)	$swap(22)$
	(v_7, v_3)	11	(v_3, v_4)	$swap(12)$
	(v_7, v_3)	21	(v_3, v_8)	$swap(22)$
τ_{v_4}	(v_3, v_4)	12	out_1	pop
τ_{v_5}	(v_2, v_5)	40	(v_5, v_6)	pop
τ_{v_6}	(v_2, v_6)	30	(v_6, v_7)	$swap(31)$
	(v_5, v_6)	30	(v_6, v_7)	$swap(31)$
	(v_5, v_6)	61	(v_6, v_7)	$swap(62)$
	(v_5, v_6)	71	(v_6, v_7)	$swap(72)$
τ_{v_7}	(v_6, v_7)	31	(v_7, v_3)	pop
	(v_6, v_7)	62	(v_7, v_3)	$swap(11)$
	(v_6, v_7)	72	(v_7, v_8)	$swap(22)$
τ_{v_8}	(v_3, v_8)	22	out_2	pop
	(v_7, v_8)	22	out_2	pop

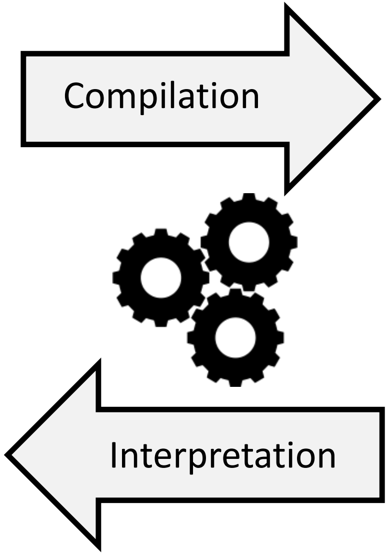




local FFT	Out-I	In-Label	Out-I	op
τ_{v_2}	(v_2, v_3)	11	(v_2, v_6)	$push(30)$
	(v_2, v_3)	21	(v_2, v_6)	$push(30)$
	(v_2, v_6)	30	(v_2, v_5)	$push(40)$

global FFT	Out-I	In-Label	Out-I	op
τ'_{v_2}	(v_2, v_3)	11	(v_2, v_6)	$swap(61)$
	(v_2, v_3)	21	(v_2, v_6)	$swap(71)$
	(v_2, v_6)	61	(v_2, v_5)	$push(40)$
	(v_2, v_6)	71	(v_2, v_5)	$push(40)$

MPLS **configurations**, Segment Routing etc.



Compilation

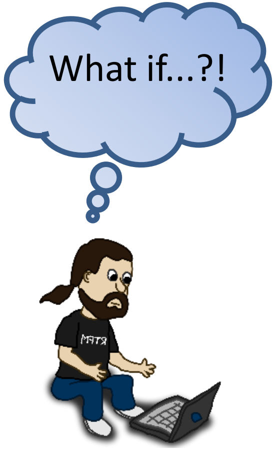
Interpretation

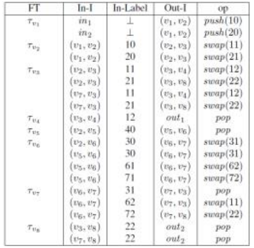
$$\mathrm{pX} \Rightarrow \mathrm{qXX}$$
$$pX \Rightarrow qYX$$
$$qY \Rightarrow rYY$$
$$rY \Rightarrow r$$
$$rX \Rightarrow pX$$

Pushdown Automaton and **Prefix Rewriting Systems** Theory

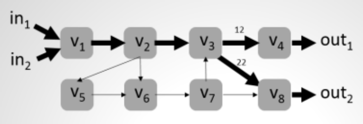
Polynomial-Time Verification: An Automata-Theoretic Approach

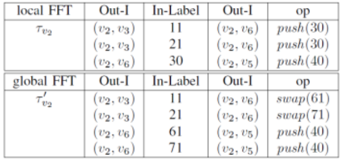
Use cases: Sysadmin ***issues queries*** to test certain properties, or do it on a ***regular basis*** automatically!





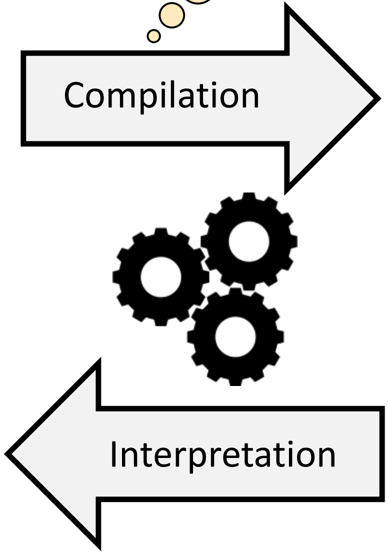
FT	In-I	In-Label	Out-I	op
τ_{v_1}	in_1	$\perp$	(v_1, v_2)	$push(10)$
	in_2	$\perp$	(v_1, v_2)	$push(20)$
τ_{v_2}	(v_1, v_2)	10	(v_2, v_3)	$swap(11)$
	(v_1, v_2)	20	(v_2, v_3)	$swap(21)$
τ_{v_3}	(v_2, v_3)	11	(v_3, v_4)	$swap(12)$
	(v_2, v_3)	21	(v_3, v_8)	$swap(22)$
	(v_7, v_3)	11	(v_3, v_4)	$swap(12)$
	(v_7, v_3)	21	(v_3, v_8)	$swap(22)$
τ_{v_4}	(v_3, v_4)	12	out_1	pop
τ_{v_5}	(v_2, v_5)	40	(v_5, v_6)	pop
τ_{v_6}	(v_2, v_6)	30	(v_6, v_7)	$swap(31)$
	(v_5, v_6)	30	(v_6, v_7)	$swap(31)$
	(v_5, v_6)	61	(v_6, v_7)	$swap(62)$
	(v_5, v_6)	71	(v_6, v_7)	$swap(72)$
τ_{v_7}	(v_6, v_7)	31	(v_7, v_3)	pop
	(v_6, v_7)	62	(v_7, v_3)	$swap(11)$
	(v_6, v_7)	72	(v_7, v_8)	$swap(22)$
τ_{v_8}	(v_3, v_8)	22	out_2	pop
	(v_7, v_8)	22	out_2	pop





local FFT	Out-I	In-Label	Out-I	op
τ_{v_2}	(v_2, v_3)	11	(v_2, v_6)	$push(30)$
	(v_2, v_3)	21	(v_2, v_6)	$push(30)$
	(v_2, v_6)	30	(v_2, v_5)	$push(40)$
global FFT	**Out-I**	**In-Label**	**Out-I**	**op**
τ'_{v_2}	(v_2, v_3)	11	(v_2, v_6)	$swap(61)$
	(v_2, v_3)	21	(v_2, v_6)	$swap(71)$
	(v_2, v_6)	61	(v_2, v_5)	$push(40)$
	(v_2, v_6)	71	(v_2, v_5)	$push(40)$

MPLS **configurations**, Segment Routing etc.



Compilation

Interpretation

$$pX \Rightarrow qXX$$

$$pX \Rightarrow qYX$$

$$qY \Rightarrow rYY$$

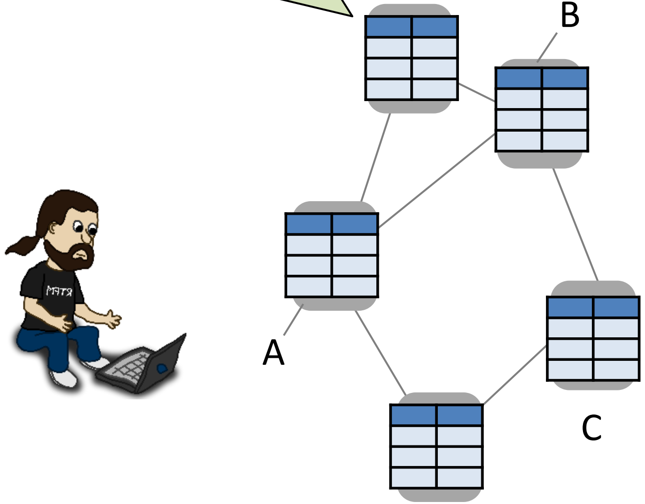
$$rY \Rightarrow r$$

$$rX \Rightarrow pX$$

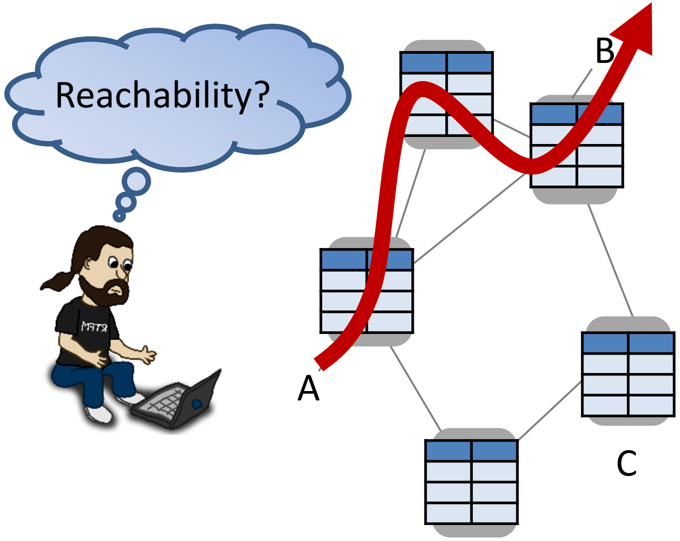
Pushdown Automaton and **Prefix Rewriting Systems** Theory

Responsibilities of a Sysadmin

Routers and switches store list of forwarding rules, and conditional failover rules.



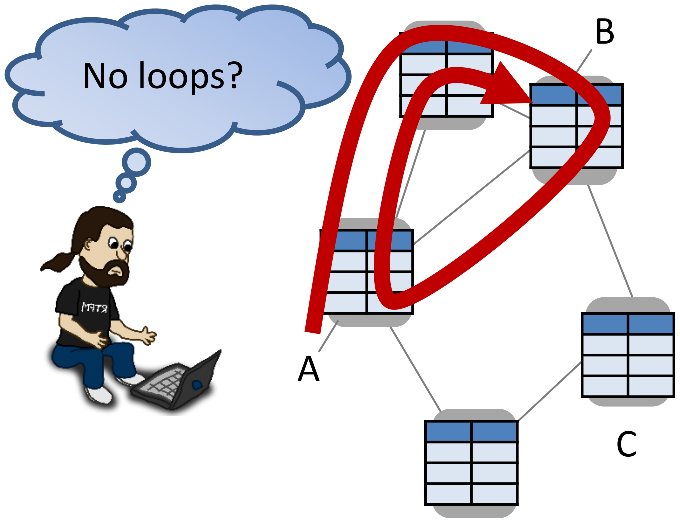
Responsibilities of a Sysadmin



Sysadmin responsible for:

- **Reachability:** Can traffic from ingress port A reach egress port B?

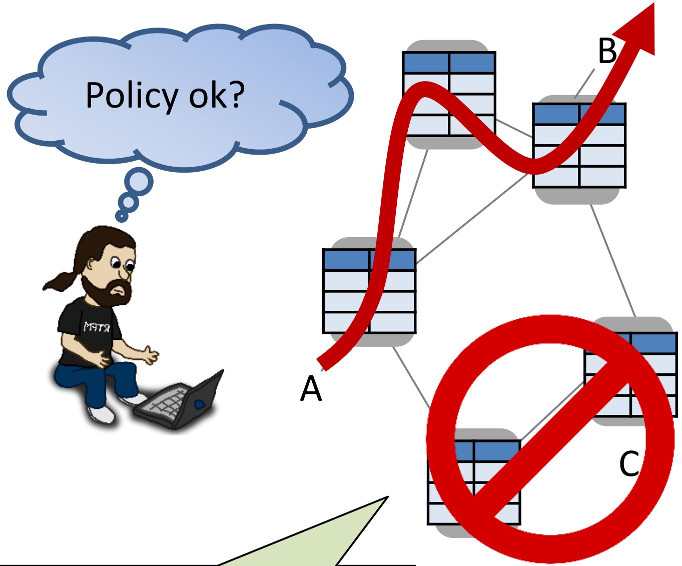
Responsibilities of a Sysadmin



Sysadmin responsible for:

- **Reachability:** Can traffic from ingress port A reach egress port B?
- **Loop-freedom:** Are the routes implied by the forwarding rules loop-free?

Responsibilities of a Sysadmin

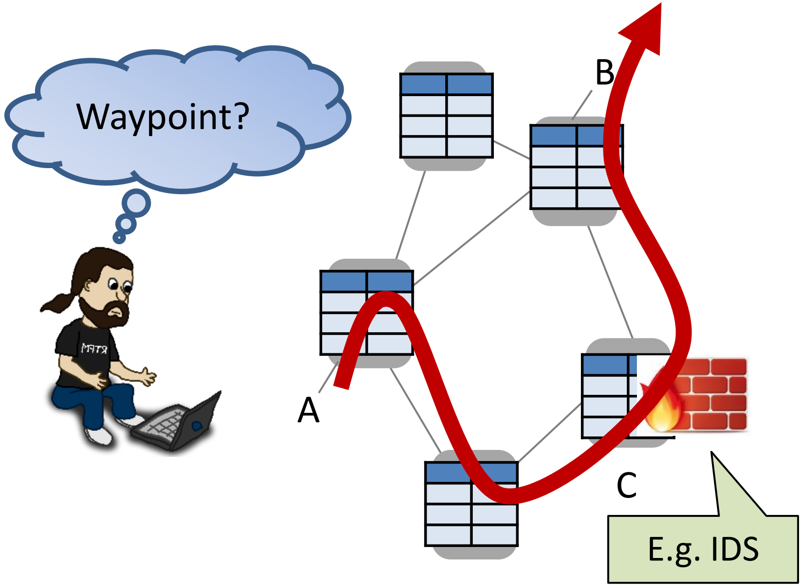


E.g. ***NORDUnet***: no traffic via Iceland (expensive!).

Sysadmin responsible for:

- **Reachability:** Can traffic from ingress port A reach egress port B?
- **Loop-freedom:** Are the routes implied by the forwarding rules loop-free?
- **Policy:** Is it ensured that traffic from A to B never goes via C?

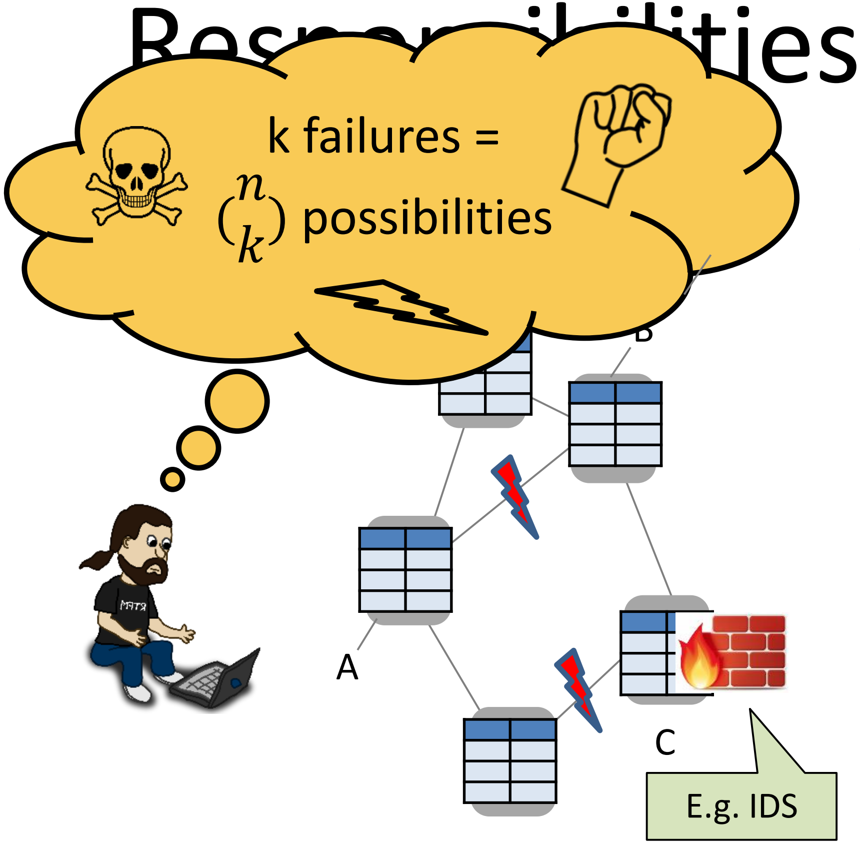
Responsibilities of a Sysadmin



Sysadmin responsible for:

- **Reachability:** Can traffic from ingress port A reach egress port B?
- **Loop-freedom:** Are the routes implied by the forwarding rules loop-free?
- **Policy:** Is it ensured that traffic from A to B never goes via C?
- **Waypoint enforcement:** Is it ensured that traffic from A to B is always routed via a node C (e.g., intrusion detection system or a firewall)?

Responsibilities of a Sysadmin



k failures = $\binom{n}{k}$ possibilities

<span style="color:blue">Sysadmin</span> responsible for:

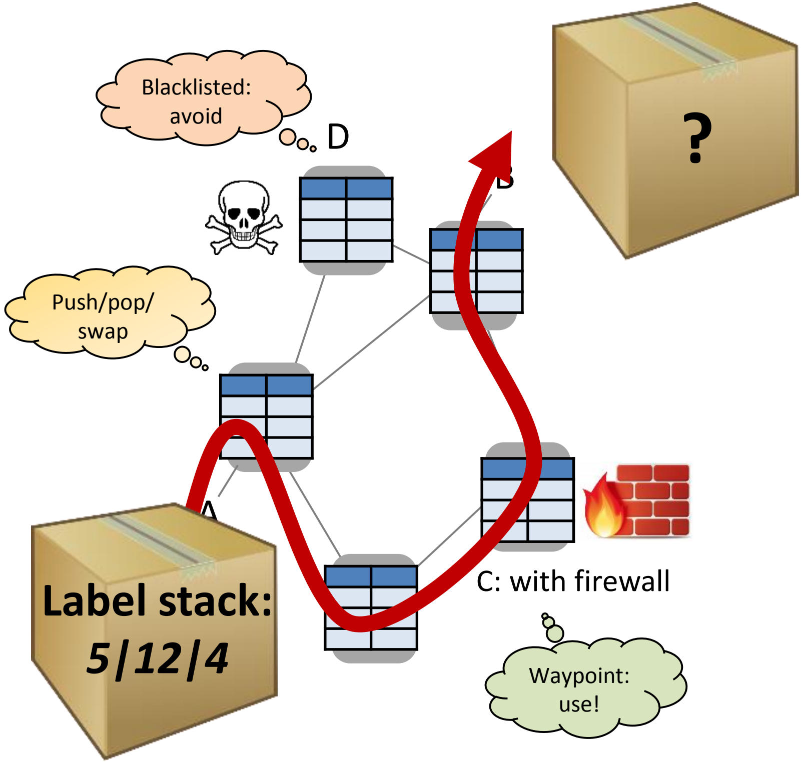
- **Reachability:** Can traffic from ingress port A reach egress port B?
- **Loop-freedom:** Are the routes implied by the forwarding rules loop-free?
- **Policy:** Is it ensured that traffic from A to B never goes via C?
- **Waypoint enforcement:** Is it ensured that traffic from A to B is always routed via a node C (e.g., intrusion detection system or a firewall)?

… and everything even under multiple failures?!

Queries May Also Depend on Header

Interface Connectivity Problem

- Can a packet arriving at A ***with header h*** reach B? (Similar for our other properties.)



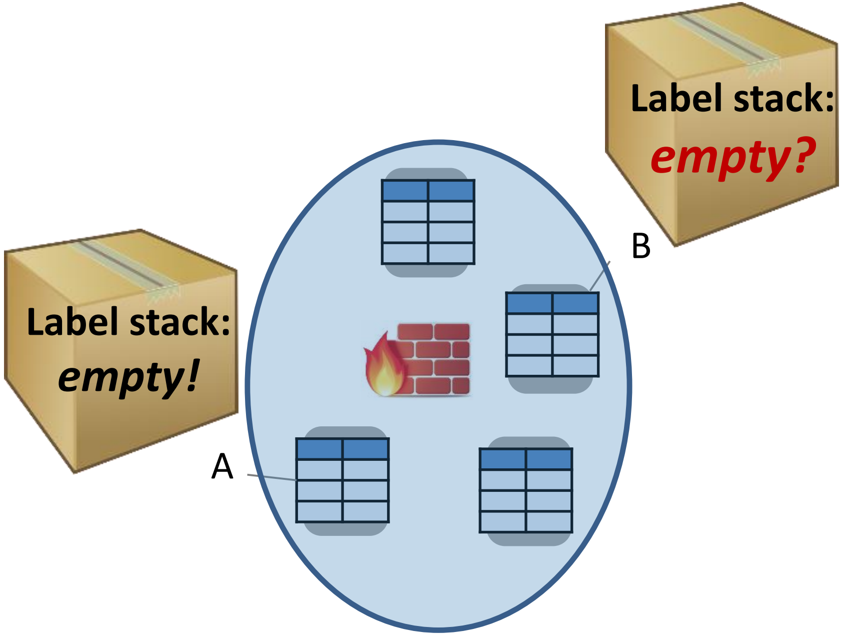
Queries May Also Depend on Header

Interface Connectivity Problem

- Can a packet arriving at A ***with header h*** reach B? (Similar for our other properties.)

Transparency

- MPLS: ***transit networks***!
- Will all packets arriving with empty header at A leave at B also with the ***empty header***?



Queries May Also Depend on Header

Interface Connectivity Problem

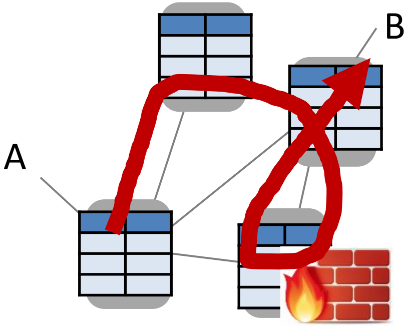
- Can a packet arriving at A ***with header h*** reach B? (Similar for our other properties.)

Transparency

- MPLS: ***transit networks***!
- Will all packets arriving with empty header at A leave at B also with the ***empty header***?

Cyclic and repeated routing

- Will a packet traverse some node ***more than r-times***?
- And what is the ***max stack size*** that a packet may have?



Waypoints and service chains: result in (a small number of) loops!

A Network Model

- Network: a 7-tuple

$$N = (V, E, I_v^{in}, I_v^{out}, \lambda_v, L, \delta_v^F)$$

Nodes

Links

Incoming interfaces

Outgoing interfaces

Set of labels in packet header

A Network Model

- Network: a 7-tuple

$$N = (V, E, I_v^{in}, I_v^{out}, \lambda_v, L, \delta_v^F)$$

Interface function

Interface function: maps outgoing interface to next hop node and incoming interface to previous hop node

$$\lambda_v : I_v^{in} \cup I_v^{out} \rightarrow V$$

That is: $(\lambda_v(in), v) \in E$ and $(v, \lambda_v(out)) \in E$

A Network Model

- Network: a 7-tuple

$$N = (V, E, I_v^{in}, I_v^{out}, \lambda_v, L, \delta_v^F)$$

Routing function

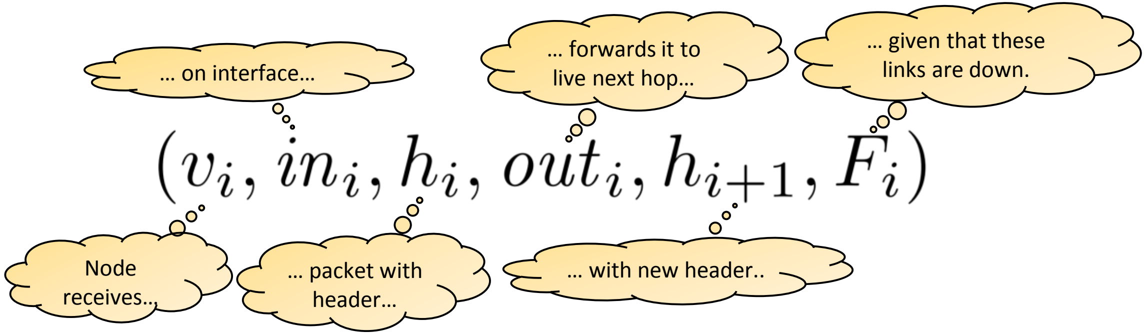
Routing function: for each set of failed links $F \subseteq E$, the routing function

$$\delta_v^F : I_v^{in} \times L^* \rightarrow 2^{(I^{out} \times L^*)}$$

defines, for all incoming interfaces and packet headers, outgoing interfaces together with modified headers.

Routing in Network

Packet routing sequence can be represented using sequence of tuples:



$$(v_i, in_i, h_i, out_i, h_{i+1}, F_i)$$

Node receives...

... on interface...

... packet with header...

... forwards it to live next hop...

... with new header..

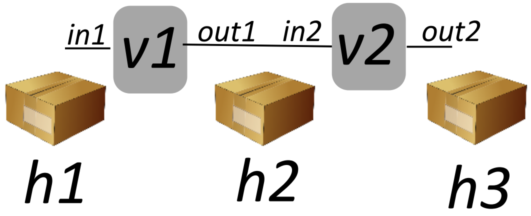
... given that these links are down.

- Example: **routing** (in)finite sequence of tuples

$$(v_1, in_1, h_1, out_1, h_2, F_1),$$

$$(v_2, in_2, h_2, out_2, h_3, F_2),$$

$$\ldots$$



MPLS Network Model

- MPLS supports **three specific operations** on header sequences:

$$Op = \{swap(\ell) \mid \ell \in L\} \cup \{push(\ell) \mid \ell \in L\} \cup \{pop\}$$

- The **local routing table** can then be defined as

$$\tau_v : I_v^{in} \times (L \cup \{\bot\}) \hookrightarrow I_v^{out} \times Op$$

Interface + label

Maps to next hop and operation

- Local **link protection function** defines backup interface

$$\pi_v : I_v^{out} \times (L \cup \{\bot\}) \hookrightarrow I_v^{out} \times Op$$

protected

backup

typically: push

MPLS Prefix Rewriting System

- Prefix rewriting system is set of **rewriting rules** $R \subseteq \Gamma^* \times \Gamma^*$
  - We will write $v \rightarrow w$ for $(v, w) \in R$:

Replace prefix

- **Prefix rewriting** rules: $vt \rightarrow_R wt$ for $t \in \Gamma^*$

  generate a **transition system** $G_R = (\Gamma^*, \rightarrow_R)$

- We call a prefix rewriting system **pushdown system** if $|v| = 2$ and $1 \leq |w| \leq 3$ for all $(v, w) \in R$

Second symbol of v: top of stack label.

First symbol of v and w: control state of pushdown system.

$|w| = 1$ pop
$|w| = 2$ swap
$|w| = 3$ push

MPLS Prefix Rewriting System

- **Control states**: (v, in) and (v, out, i)

Node and incoming link

How many times have we tried to ***reroute*** at this node already?

- **Labels**: stack symbols and $\perp$ at bottom

- Packet with header h arriving at interface *in* at v represented as **pushdown configuration**: $(v, in)h\perp$

- Packet to be forwarded at node v to outgoing interface out represented by **configuration**: $(v, out, i)h\perp$

Example Rules:
Regular Forwarding on Top-Most Label

Push label on stack

Push:

$$(v, in)\ell \to (v, out, 0)\ell'\ell \text{ if } \tau_v(in, \ell) = (out, push(\ell'))$$

Swap top of stack

Swap:

$$(v, in)\ell \to (v, out, 0)\ell' \text{ if } \tau_v(in, \ell) = (out, swap(\ell'))$$

Pop top of stack

Pop:

$$(v, in)\ell \to (v, out, 0) \text{ if } \tau_v(in, \ell) = (out, pop)$$

Example *Failover* Rules

Emumerate all rerouting options

Failover-Push:

$$(v, out, i)\ell \rightarrow (v, out', i+1)\ell'\ell \text{ for every } i,\ 0 \leq i < k,$$
$$\text{where } \pi_v(out, \ell) = (out', push(\ell'))$$

Failover-Swap:

$$(v, out, i)\ell \rightarrow (v, out', i+1)\ell' \text{ for every } i,\ 0 \leq i < k,$$
$$\text{where } \pi_v(out, \ell) = (out', swap(\ell')),$$

Failover-Pop:

$$(v, out, i)\ell \rightarrow (v, out', i+1) \text{ for every } i,\ 0 \leq i < k,$$
$$\text{where } \pi_v(out, \ell) = (out', pop).$$

Example rewriting sequence:

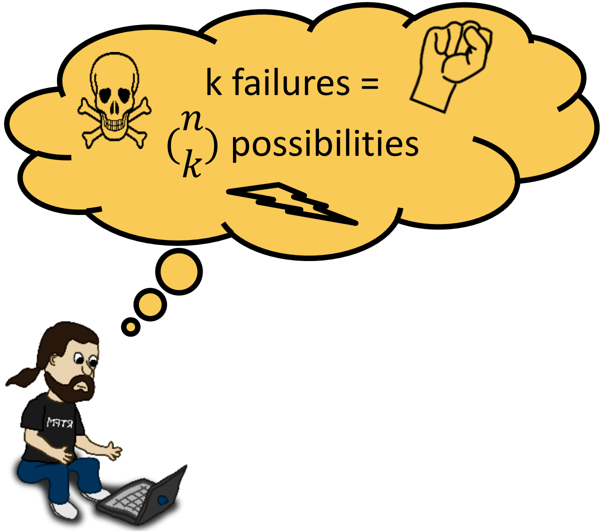
$$(v_1, in_1)h_1\bot \rightarrow (v_1, out, 0)h\bot \rightarrow (v_1, out', 1)h'\bot \rightarrow (v_1, out'', 2)h''\bot \rightarrow \ldots \rightarrow (v_1, out_1, i)h_2\bot$$

Try default

Try first backup

Try second backup

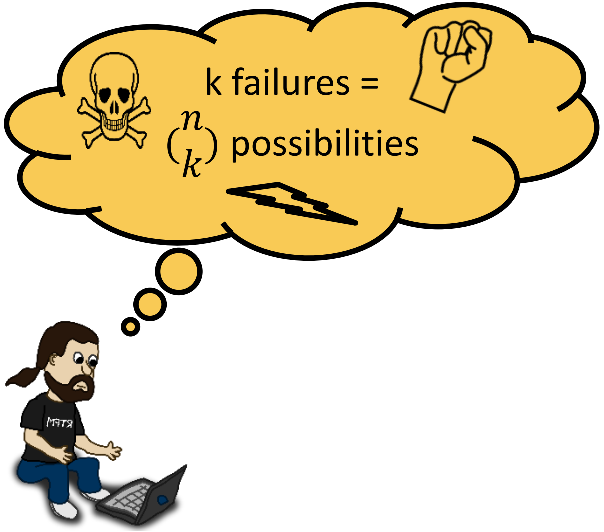
Why Polynomial Time?!



- Arbitrary number k of failures: How can I avoid checking all $\binom{n}{k}$ many options?!

- Even if we reduce to **push-down automaton**: simple operations such as emptiness testing or intersection on Push-Down Automata (PDA) is computationally non-trivial and sometimes even **undecidable**!

Why Polynomial Time?!



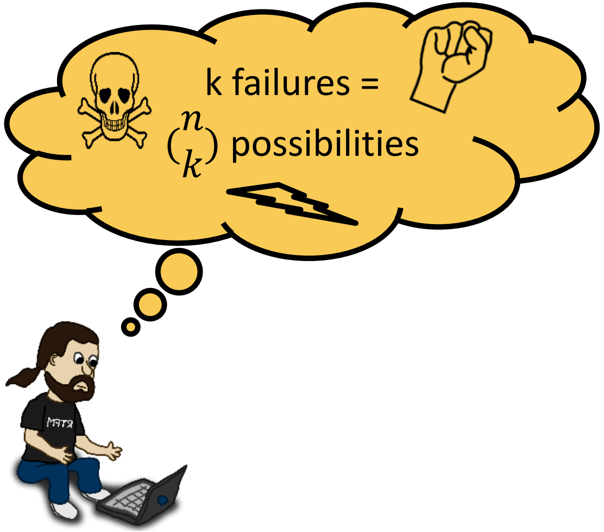
k failures = $\binom{n}{k}$ possibilities

- Arbitrary number k of failures: How can I avoid checking all $\binom{n}{k}$ many options?!

- Even if we reduce to **push-down automaton**: simple operations such as emptiness testing or intersection on Push-Down Automata (PDA) is computationally non-trivial and sometimes even **undecidable**!

This is ***not*** how we will use the PDA!

Why Polynomial Time?!



k failures = $\binom{n}{k}$ possibilities

- Arbitrary number k of failures: How can I avoid checking all $\binom{n}{k}$ many options?!
- Even if we reduce to **push-down automaton**: simple operations such as emptiness testing or intersection on Push-Down Automata (PDA) is computationally non-trivial and sometimes even **undecidable**!

The words in our language are sequences of pushdown stack symbols, not the labels of transitions.

Time for Automata Theory!

- Classic result by **Büchi** 1964: the set of all reachable configurations of a pushdown automaton a is regular set
- Hence, we can operate only on Nondeterministic Finite Automata (NFAs) when reasoning about the pushdown automata
- The resulting **regular operations** are all polynomial time
- Important result of **model checking**

Julius Richard Büchi

1924-1984

Swiss logician

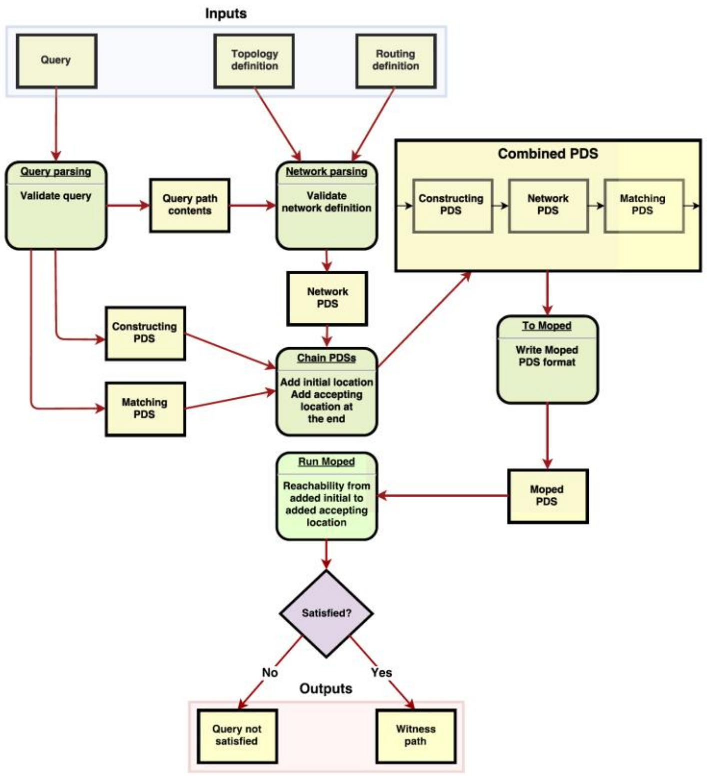
Preliminary Tool and Query Language

Part 1: Parses query and constructs Push-Down System (PDS)

- In Python 3

Part 2: Reachability analysis of constructed PDS

- Using ***Moped*** tool



query processing flow

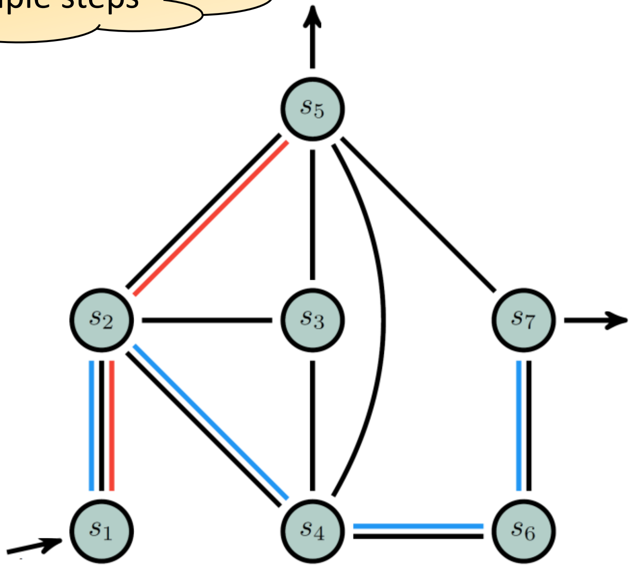
Example 1: Reachability

Question: Beginning with an empty header [], can we get from s1 to s7 in any number of steps, and end with an empty header []?

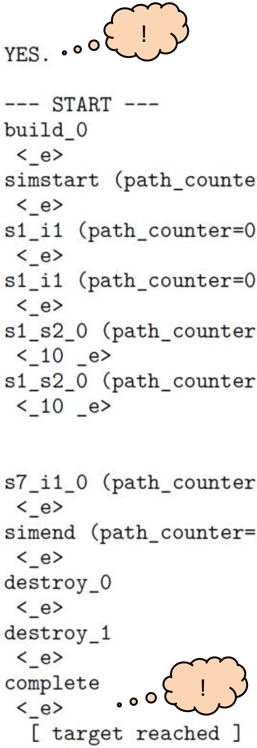
Take multiple steps

Query: []s1 >> s7[]

Empty header



Output: Yes and witness trace (excerpt)



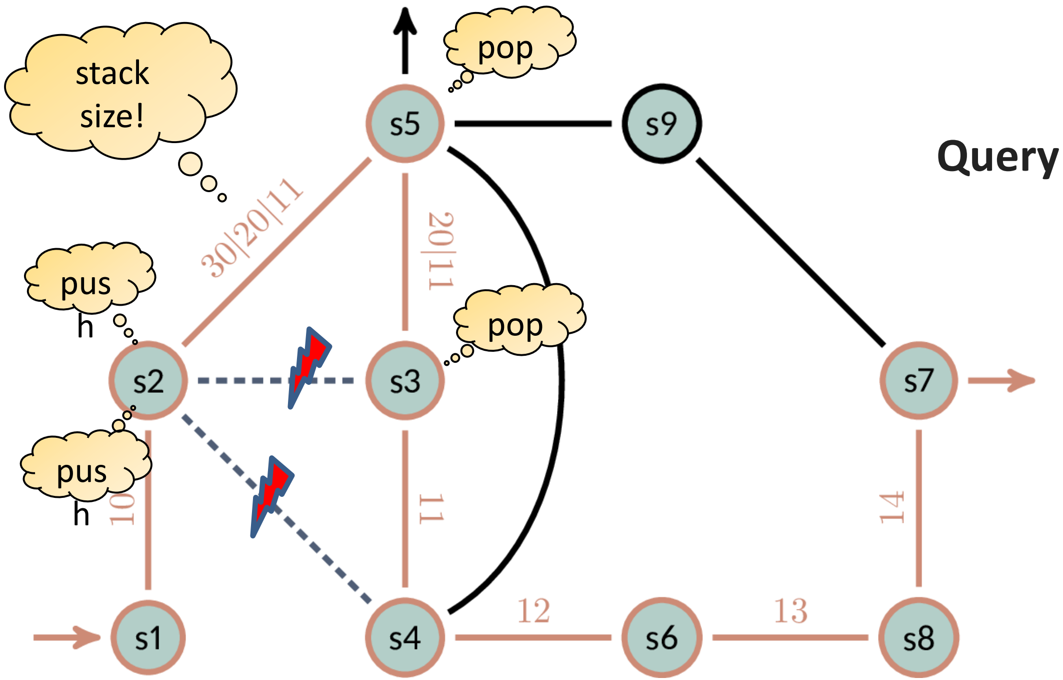
```
YES.

--- START ---
build_0
 <_e>
simstart (path_counter=0)
 <_e>
s1_i1 (path_counter=0)
 <_e>
s1_i1 (path_counter=0)
 <_e>
s1_s2_0 (path_counter=0)
 <_10 _e>
s1_s2_0 (path_counter=1)
 <_10 _e>


s7_i1_0 (path_counter=2)
 <_e>
simend (path_counter=0)
 <_e>
destroy_0
 <_e>
destroy_1
 <_e>
complete
 <_e>
  [ target reached ]
```

Example 2: Traversal With 2 Failures

Traversal test with k=2: Can traffic starting with [] go through s5, under up to k=2 failures?

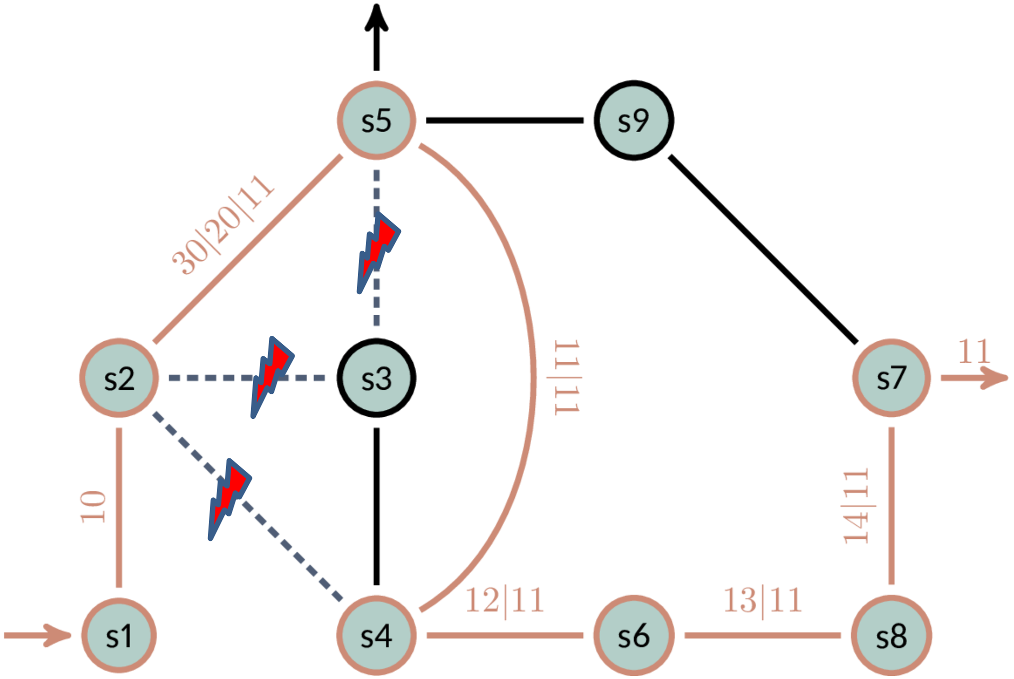


Query: k=2 [] s1 >> s5 >> s7 []

YES!

Example 3: Transparency Violation

Transparency with k=3: Can transparency be violated under up to k=3 failures?



Query: k=3 [] s1 >> s7 [+]

3 failures

empty

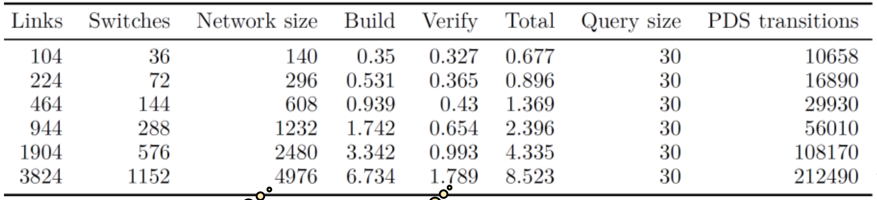
non-empty

YES!

Root cause is a misconfiguration in s5, causing it to swap to 11 instead of popping when doing the failover on s5-s4.

Preliminary Evaluation

For small queries fast: 1000s of links, within seconds

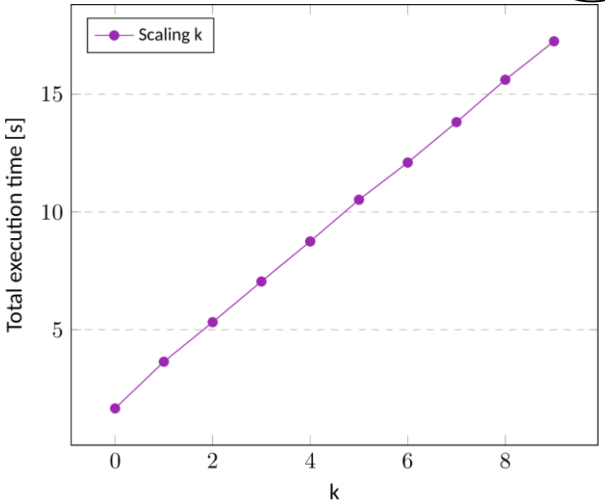


Links	Switches	Network size	Build	Verify	Total	Query size	PDS transitions
104	36	140	0.35	0.327	0.677	30	10658
224	72	296	0.531	0.365	0.896	30	16890
464	144	608	0.939	0.43	1.369	30	29930
944	288	1232	1.742	0.654	2.396	30	56010
1904	576	2480	3.342	0.993	4.335	30	108170
3824	1152	4976	6.734	1.789	8.523	30	212490

100,000s

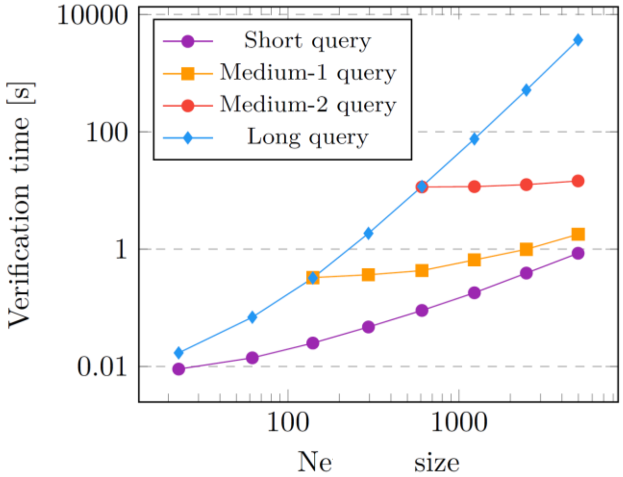
1000s

secs



# failures affects performance only linearly!

Bottleneck are large queries



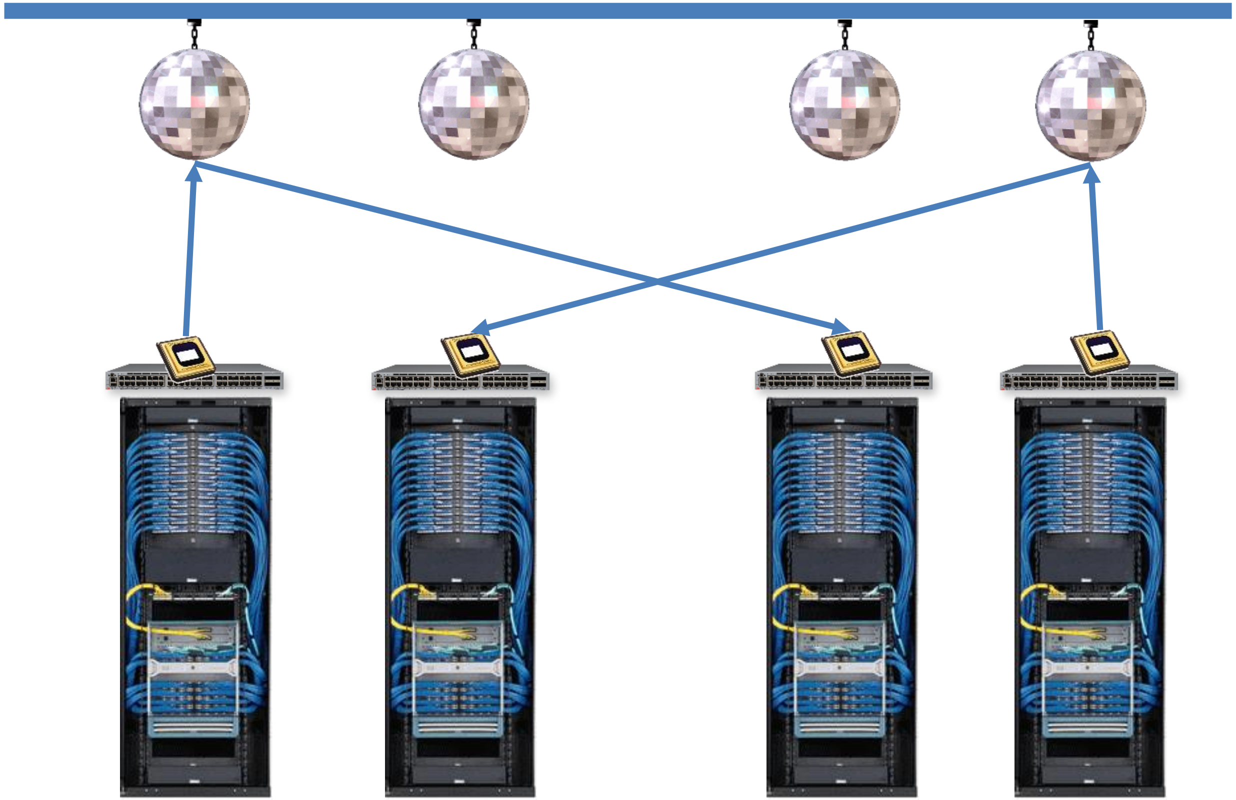
Summary of Contributions

- **Polynomial-time verification** of MPLS reachability and policy-related properties like waypointing
  - For arbitrary number of failures (up to linear in n)!
  - Supports arbitrary header sizes („infinite“)
  - Also allows to compute headers which do (not) fulfill a property
  - Allows to support a constant number of stateful nodes as well
  - Extends to Segment Routing networks based on MPLS (SR-MPLS)

- Leveraging theory from **Prefix Rewriting Systems** and **Büchi**‘s classic result

The Next Frontier of Flexibility: The Network Topology

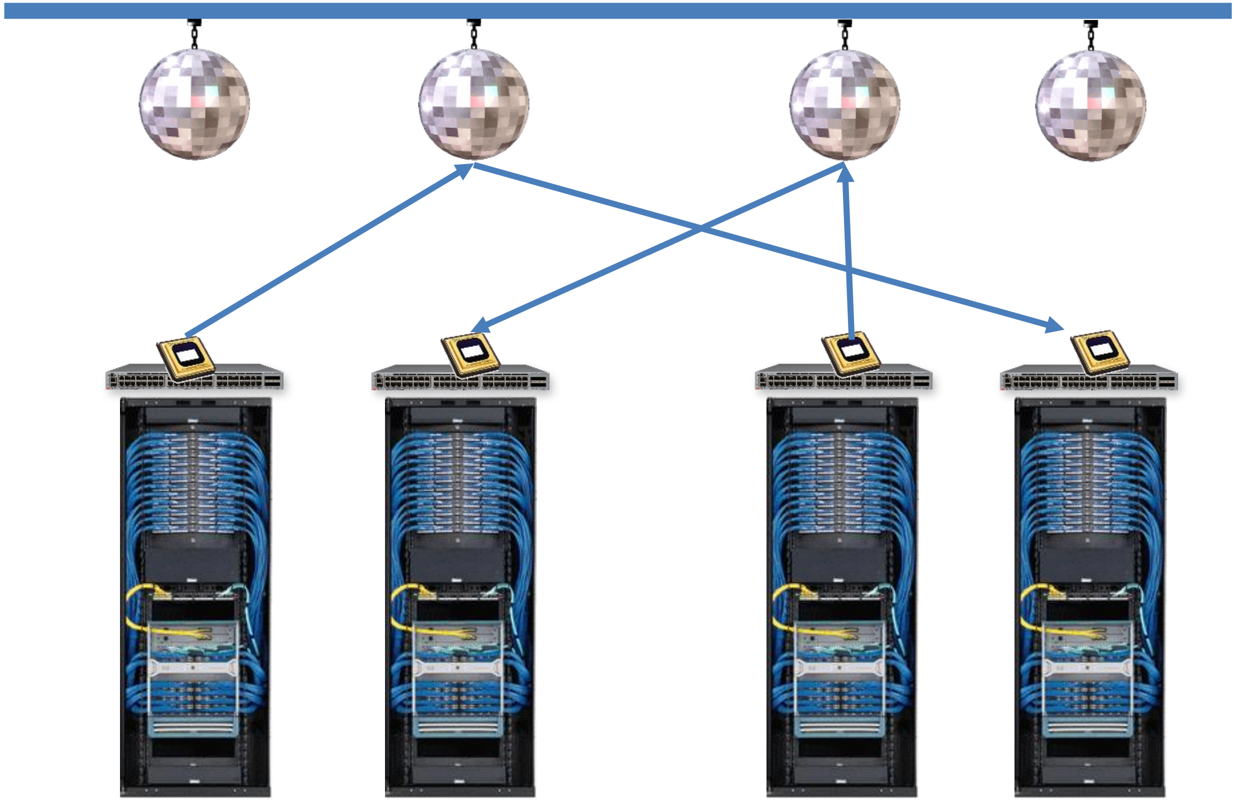
Started as a **theoretical project**, *but then*:



t=1

The Next Frontier of Flexibility: The Network Topology

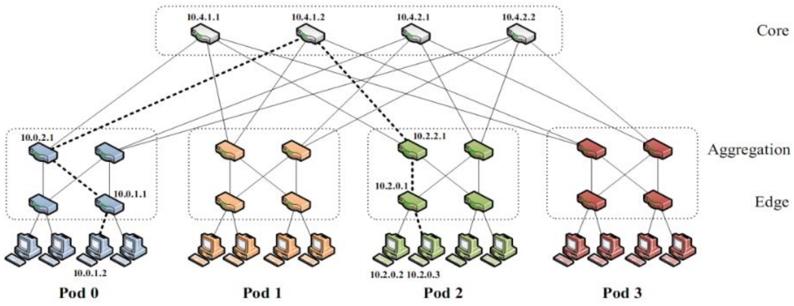
Started as a **theoretical project**, *but then*:



t=2

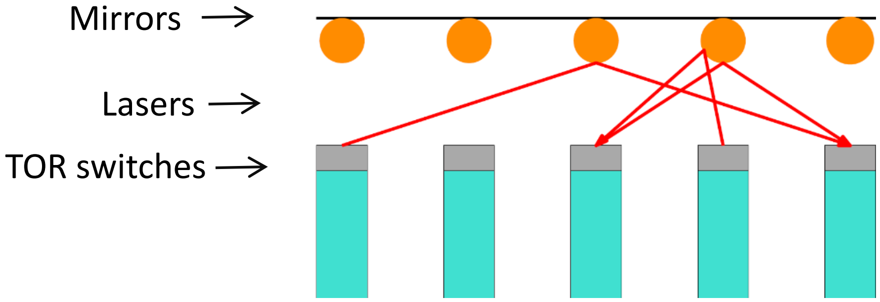
Traditional Networks: Static

- Lower bounds and undesirable **trade-offs**, e.g., degree vs diameter
- Usually optimized for the "worst-case" (**all-to-all** communication)
- Example, fat-tree topologies: provide **full bisection bandwidth**



Our Vision: DANs and SANs

- DAN: Demand-Aware Network
  - Statically optimized **toward the demand**
- SAN: Self-Adjusting Network
  - **Dynamically optimized toward** the (time-varying) demand



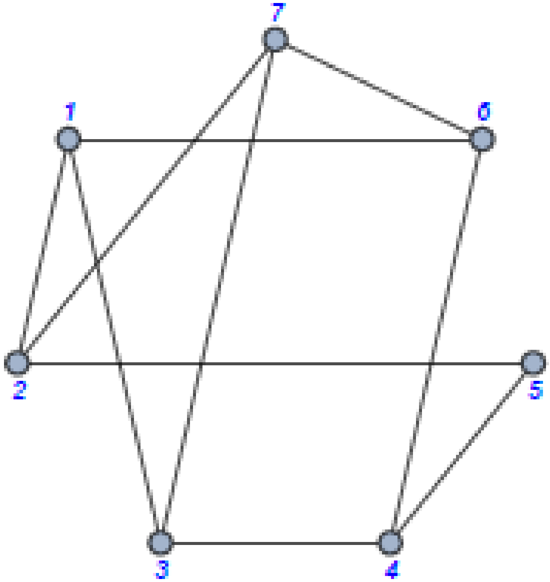
Our Research Vision: Demand-Aware Networks (DANs)

Destinations

Sources

$$
\begin{array}{c|ccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & 0 & \frac{2}{65} & \frac{1}{13} & \frac{1}{65} & \frac{1}{65} & \frac{2}{65} & \frac{3}{65} \\
2 & \frac{2}{65} & 0 & \frac{1}{65} & 0 & 0 & 0 & \frac{2}{65} \\
3 & \frac{1}{13} & \frac{1}{65} & 0 & \frac{2}{65} & 0 & 0 & \frac{1}{13} \\
4 & \frac{1}{65} & 0 & \frac{2}{65} & 0 & \frac{4}{65} & 0 & 0 \\
5 & \frac{1}{65} & 0 & \frac{3}{65} & \frac{4}{65} & 0 & 0 & 0 \\
6 & \frac{2}{65} & 0 & 0 & 0 & 0 & 0 & \frac{3}{65} \\
7 & \frac{3}{65} & \frac{2}{65} & \frac{1}{13} & 0 & 0 & \frac{3}{65} & 0 \\
\hline
\end{array}
$$

design



Demand matrix: joint distribution

DAN (of constant degree)

Our Research Vision:
Deman Networks (DANs)

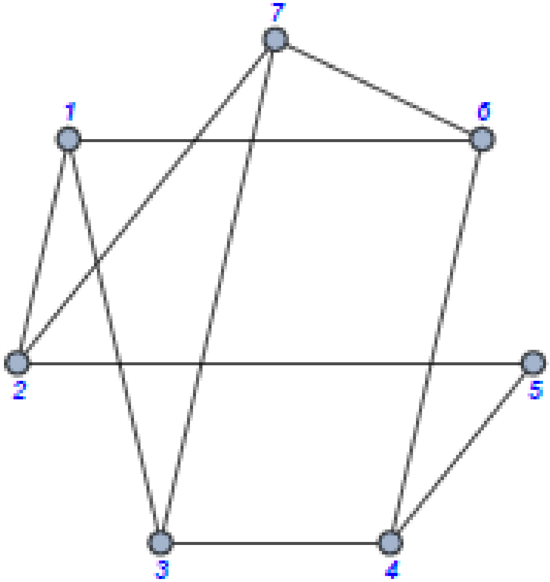
Can be seen as a graph as well: the workload!

Destinations

Sources

$$\begin{array}{c|ccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & 0 & \frac{2}{65} & \frac{1}{13} & \frac{1}{65} & \frac{1}{65} & \frac{2}{65} & \frac{3}{65} \\
2 & \frac{2}{65} & 0 & \frac{1}{65} & 0 & 0 & 0 & \frac{2}{65} \\
3 & \frac{1}{13} & \frac{1}{65} & 0 & \frac{2}{65} & 0 & 0 & \frac{1}{13} \\
4 & \frac{1}{65} & 0 & \frac{2}{65} & 0 & \frac{4}{65} & 0 & 0 \\
5 & \frac{1}{65} & 0 & \frac{3}{65} & \frac{4}{65} & 0 & 0 & 0 \\
6 & \frac{2}{65} & 0 & 0 & 0 & 0 & 0 & \frac{3}{65} \\
7 & \frac{3}{65} & \frac{2}{65} & \frac{1}{13} & 0 & 0 & \frac{3}{65} & 0
\end{array}$$

design



Demand matrix: joint distribution

DAN (of constant degree)

Our Research Vision: Demand-Aware Networks (DANs)

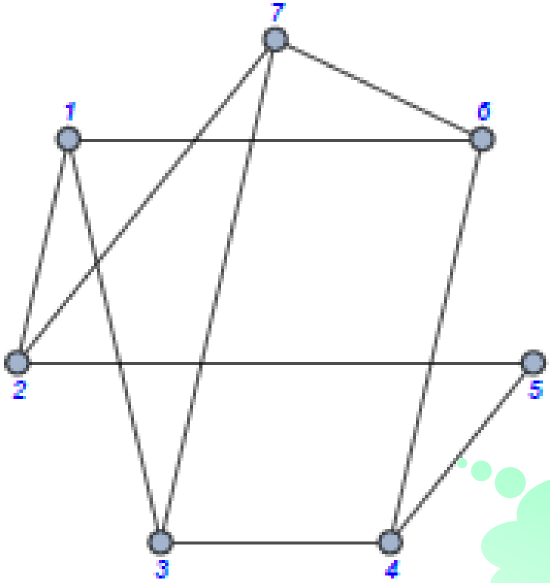
Destinations

Sources

$$\begin{array}{c|ccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & 0 & \frac{2}{65} & \frac{1}{13} & \frac{1}{65} & \frac{1}{65} & \frac{2}{65} & \frac{3}{65} \\
2 & \frac{2}{65} & 0 & \frac{1}{65} & 0 & 0 & [illegible] & [illegible] \\
3 & \frac{1}{13} & \frac{1}{65} & 0 & \frac{2}{65} & 0 & 0 & [illegible] \\
4 & \frac{1}{65} & 0 & \frac{2}{65} & 0 & \frac{4}{65} & 0 & 0 \\
5 & \frac{1}{65} & 0 & \frac{3}{65} & \frac{4}{65} & 0 & 0 & 0 \\
6 & \frac{2}{65} & 0 & 0 & 0 & 0 & 0 & \frac{3}{65} \\
7 & \frac{3}{65} & \frac{2}{65} & \frac{1}{13} & 0 & 0 & \frac{3}{65} & 0 \\
\hline
\end{array}$$

Much from 4 to 5.

design



Makes sense to add link.

Demand matrix: joint distribution

DAN (of constant degree)

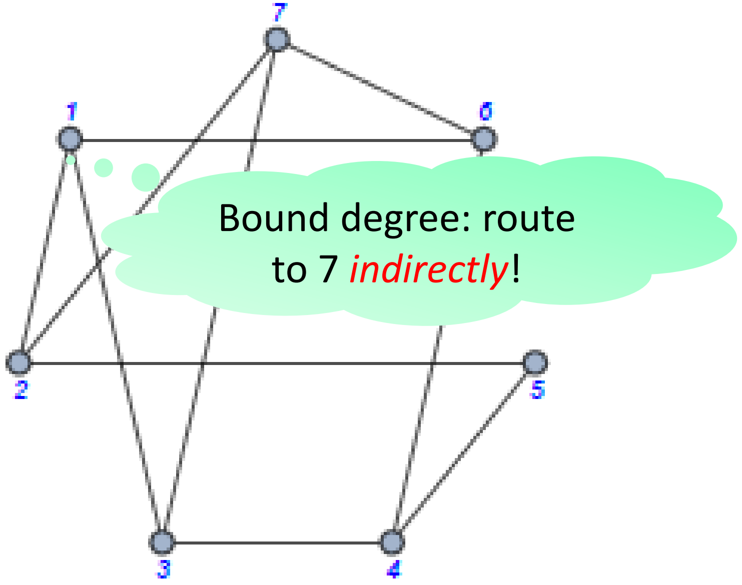
Our Research Vision:
Demand-Aware Networks (DANs)

1 communicates to many.

Sources

$$
\begin{array}{c|ccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & 0 & \frac{2}{65} & \frac{1}{13} & \frac{1}{65} & \frac{1}{65} & \frac{2}{65} & \frac{3}{65} \\
2 & \frac{2}{65} & 0 & \frac{1}{65} & 0 & 0 & 0 & \frac{2}{65} \\
3 & \frac{1}{13} & \frac{1}{65} & 0 & \frac{2}{65} & 0 & 0 & \frac{1}{13} \\
4 & \frac{1}{65} & 0 & \frac{2}{65} & 0 & \frac{4}{65} & 0 & 0 \\
5 & \frac{1}{65} & 0 & \frac{3}{65} & \frac{4}{65} & 0 & 0 & 0 \\
6 & \frac{2}{65} & 0 & 0 & 0 & 0 & 0 & \frac{3}{65} \\
7 & \frac{3}{65} & \frac{2}{65} & \frac{1}{13} & 0 & 0 & \frac{3}{65} & 0 \\
\hline
\end{array}
$$

design



Bound degree: route to 7 *indirectly*!

Demand matrix: joint distribution

DAN (of constant degree)

Our Research Vision: Demand-Aware Networks (DANs)

Destinations

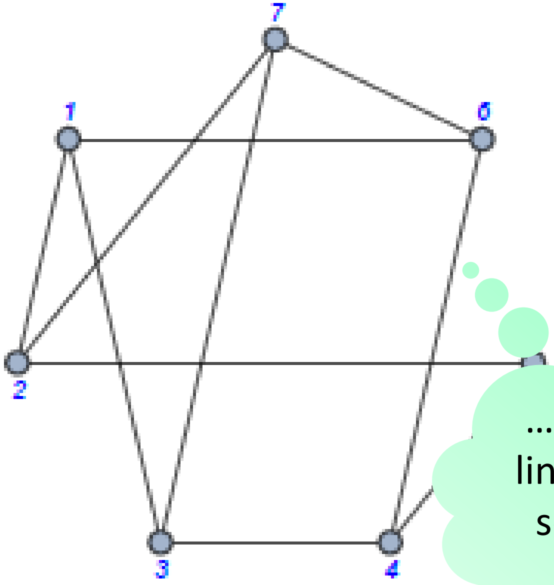
Sources

	1	2	3	4	5	6	7
1	0	$\frac{2}{65}$	$\frac{1}{13}$	$\frac{1}{65}$	$\frac{1}{65}$	$\frac{2}{65}$	$\frac{3}{65}$
2	$\frac{2}{65}$	0	$\frac{1}{65}$	0	0	0	$\frac{2}{65}$
3	$\frac{1}{13}$	$\frac{1}{65}$	0	$\frac{2}{65}$	0	0	$\frac{1}{13}$
4	$\frac{1}{65}$	0	$\frac{2}{65}$	0	$\frac{4}{65}$	0	0
5	$\frac{1}{65}$	0	$\frac{3}{65}$	$\frac{4}{65}$	0	0	0
6	$\frac{2}{65}$	0	0	0	0	0	$\frac{3}{65}$
7	$\frac{3}{65}$	$\frac{2}{65}$	$\frac{1}{13}$	0	0	3	[illegible]

4 and 6 don't communicate…

Demand matrix: joint distribution

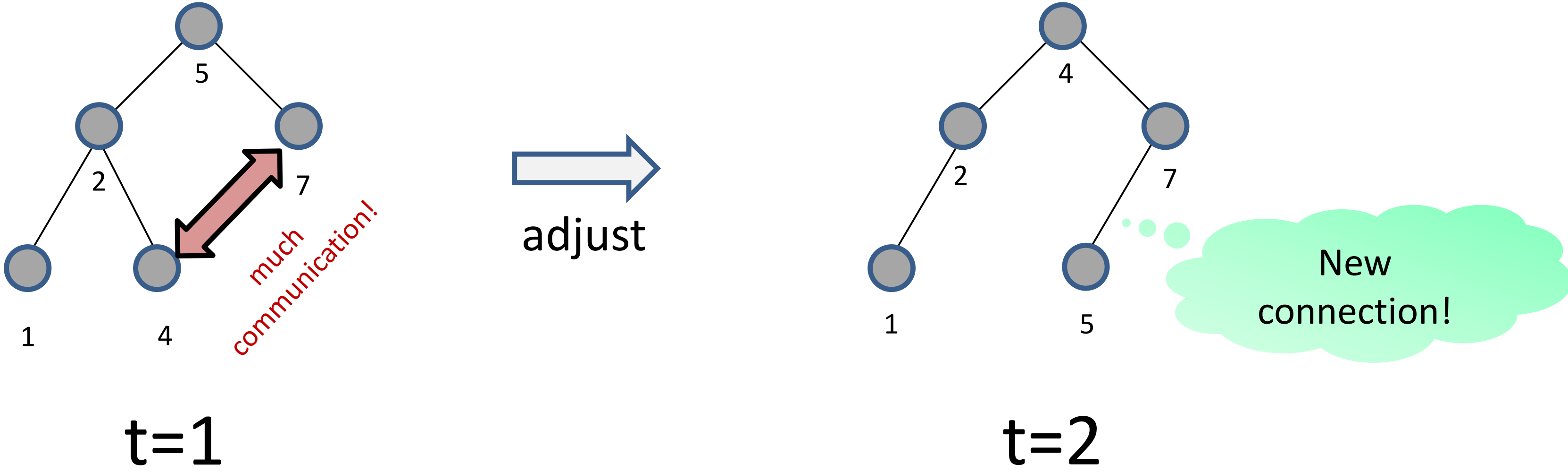
design



… but „extra“ link still makes sense: not a subgraph!

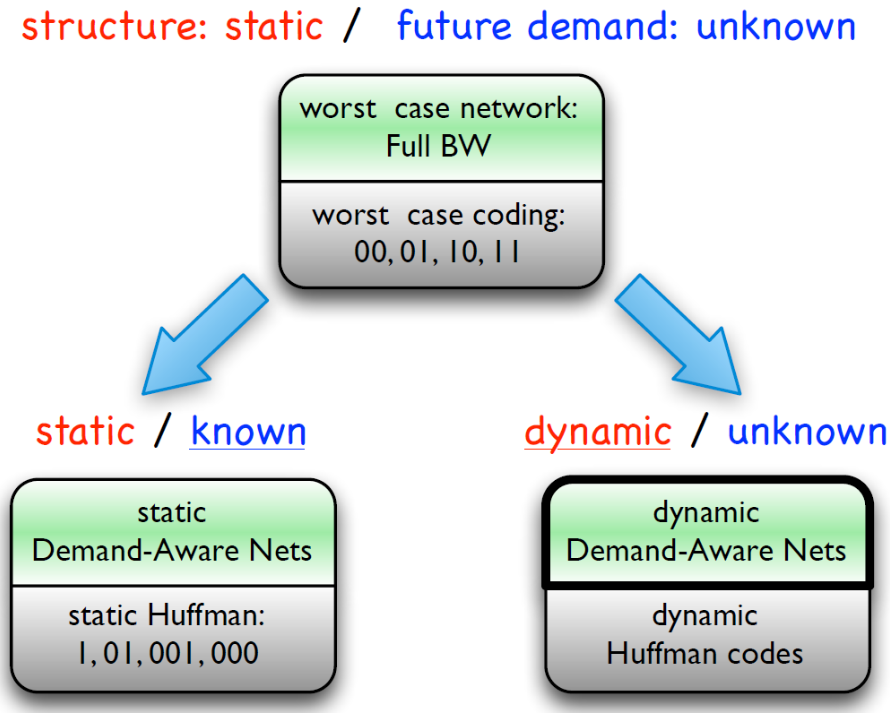
DAN (of constant degree)

Our Research Vision:
Or Even Self-Adjusting Networks (SANs)



How to **minimize reconfigurations**?
How to keep network **locally** routable?

Our Research Vision: An Analogy to Coding



structure: static / future demand: unknown

worst case network: Full BW

worst case coding: 00, 01, 10, 11

static / known

static Demand-Aware Nets

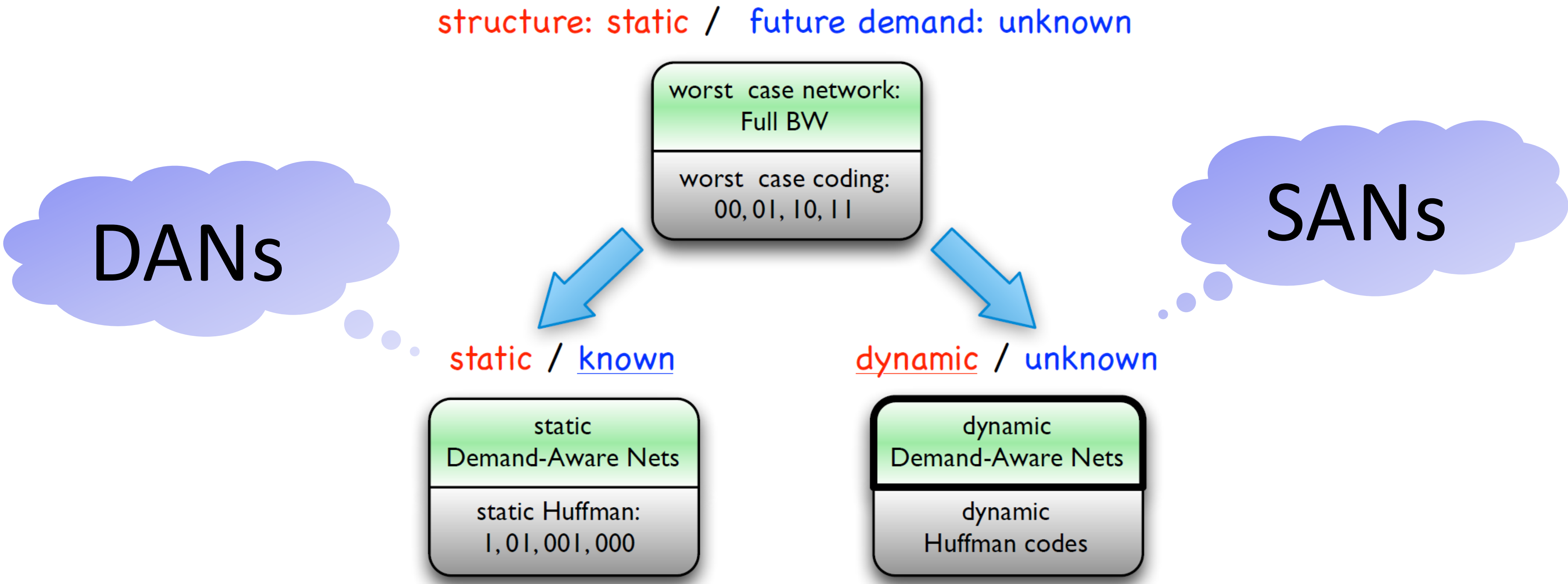
static Huffman: 1, 01, 001, 000

dynamic / unknown

dynamic Demand-Aware Nets

dynamic Huffman codes

Our Research Vision: An Analogy to Coding



structure: static / future demand: unknown

worst case network: Full BW

worst case coding: 00, 01, 10, 11

static / known

static Demand-Aware Nets

static Huffman: 1, 01, 001, 000

dynamic / unknown

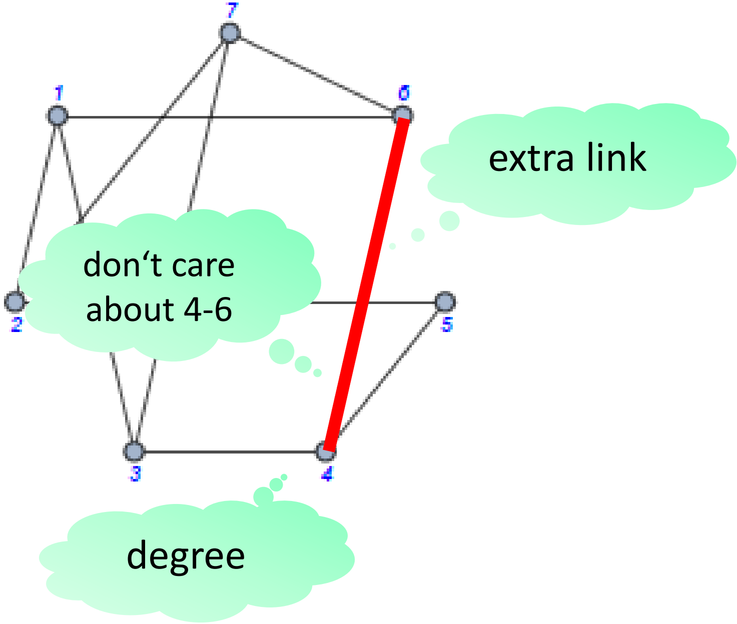
dynamic Demand-Aware Nets

dynamic Huffman codes

DAN: Relationship to…

Sparse, low-distortion **graph spanners**

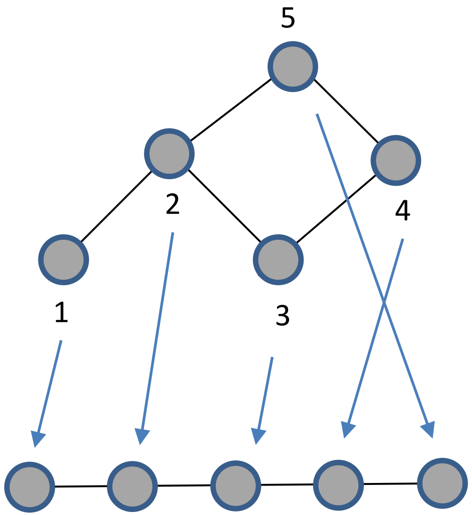
- Similar: keep distances in a „compressed network“ (few edges)
- *But:*
  - We only care about path length **between communicating nodes**, not all node pairs
  - We want **constant degree**
  - Not restricted to subgraph but can have „**additional links**“ (like geometric spanners)



DAN: Relationship to…

Minimum Linear **Arrangement** (MLA)

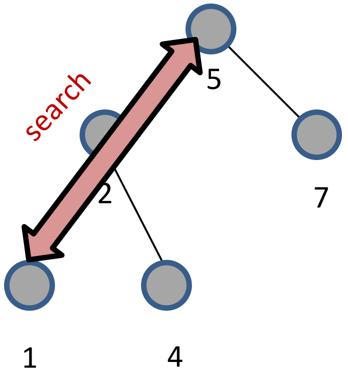
- MLA: map guest graph to line (host graph) so that sum of distances is minimal
- DAN similar: if degree bound = 2, DAN is line or ring (or sets of lines/rings)
- *But* unlike "**graph embedding problems**"
  - The host graph is also ***subject to optimization***
  - Does this render the problem simpler or harder?



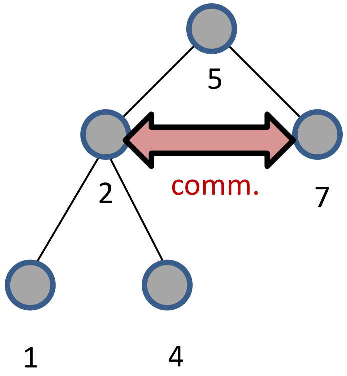
SAN: Relationship to…

- **Self-adjusting datastructures** like splay trees

- *But:* Requests are „pair-wise“, not only „from the root“



Splay Tree



SplayNet

Many interesting research questions

- How to design **static** demand-aware networks?

- How much better can demand-aware networks be compared to **demand-oblivious** networks?

- How to design **dynamic** or even **decentralized** self-adjusting demand-aware networks?

An Entropy Lower Bound

- EPL related to **entropy**. Intuition:
  - ***High entropy:*** e.g., uniform distribution, not much structure, long paths
  - ***Low entropy:*** can exploit structure to create topologies with short paths

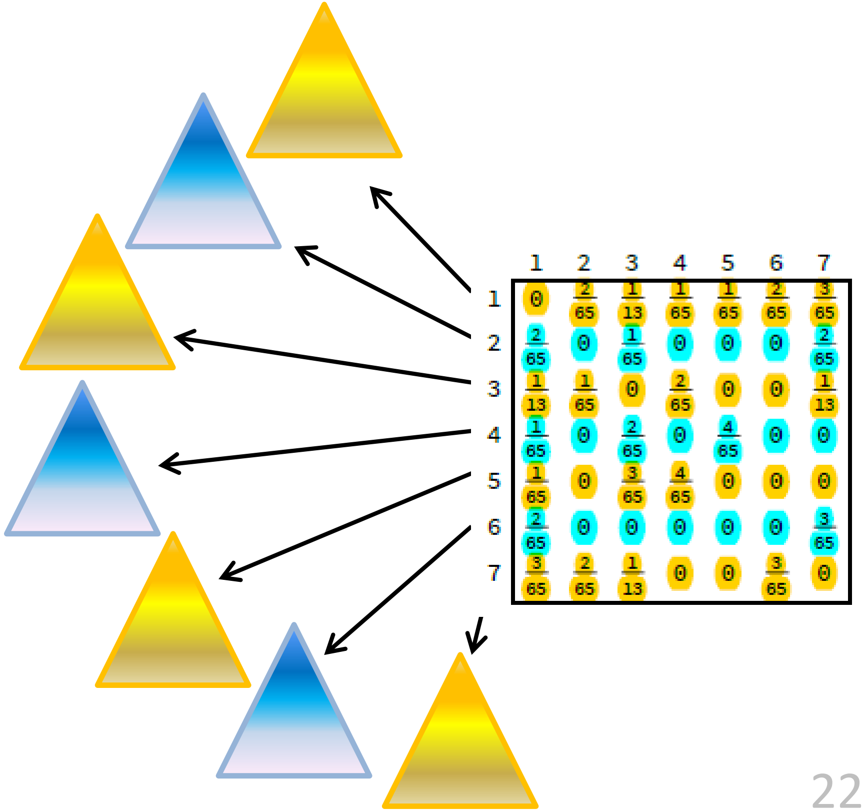
- **Theorem:** Let X, Y be the ***marginal distributions*** of the sources and destinations in demand $\mathcal{D}$ respectively. Then

$$\mathrm{EPL}(\mathcal{D}, \Delta) \geq \Omega(\mathrm{H}_\Delta(\mathrm{Y}|\mathrm{X}) + \mathrm{H}_\Delta(\mathrm{X}|\mathrm{Y}))$$

- Recall **conditional entropy**: Average uncertainty of X given Y
  - $\mathrm{H}(\mathrm{X}|\mathrm{Y}) = \sum_{\mathrm{i}=1}^{\mathrm{n}} \mathrm{p}(\mathrm{x_i},\mathrm{y_j}) \log_2(1/\mathrm{p}(\mathrm{x_i}|\mathrm{y_j}))$

Lower Bound: Idea

- **Proof idea** (EPL=$\Omega(H_\Delta(Y|X))$):
- Build optimal Δ-ary tree for each source i: entropy lower bound known on EPL known for binary trees (Mehlhorn 1975 for BST but proof does not need search property)
- Consider union of all trees
- Violates degree restriction but valid lower bound

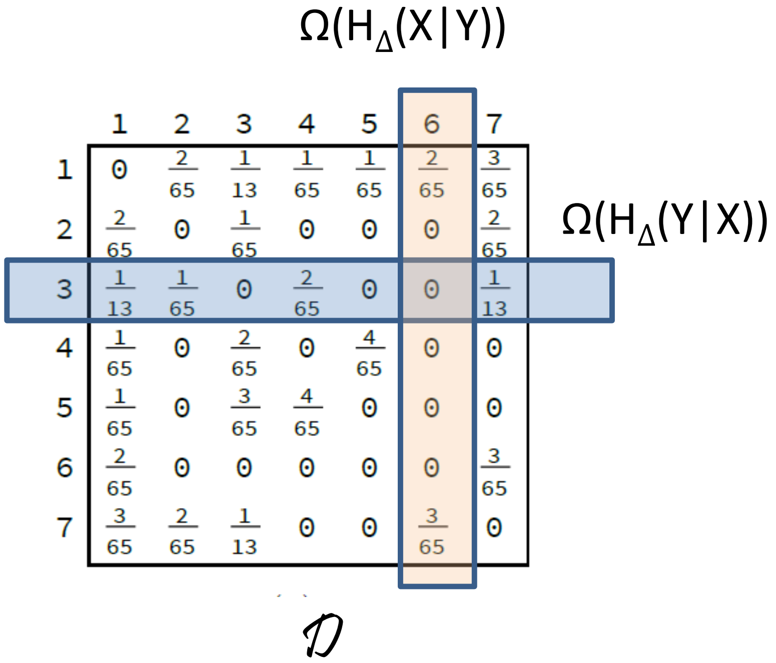


$$\begin{array}{c|ccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & 0 & \frac{2}{65} & \frac{1}{13} & \frac{1}{65} & \frac{1}{65} & \frac{2}{65} & \frac{3}{65} \\
2 & \frac{2}{65} & 0 & \frac{1}{65} & 0 & 0 & 0 & \frac{2}{65} \\
3 & \frac{1}{13} & \frac{1}{65} & 0 & \frac{2}{65} & 0 & 0 & \frac{1}{13} \\
4 & \frac{1}{65} & 0 & \frac{2}{65} & 0 & \frac{4}{65} & 0 & 0 \\
5 & \frac{1}{65} & 0 & \frac{3}{65} & \frac{4}{65} & 0 & 0 & 0 \\
6 & \frac{2}{65} & 0 & 0 & 0 & 0 & 0 & \frac{3}{65} \\
7 & \frac{3}{65} & \frac{2}{65} & \frac{1}{13} & 0 & 0 & \frac{3}{65} & 0
\end{array}$$

Lower Bound: Idea

Do this in **both dimensions**:

EPL ≥ $\Omega(\max\{H_\Delta(Y|X), H_\Delta(X|Y)\})$



$\Omega(H_\Delta(X|Y))$

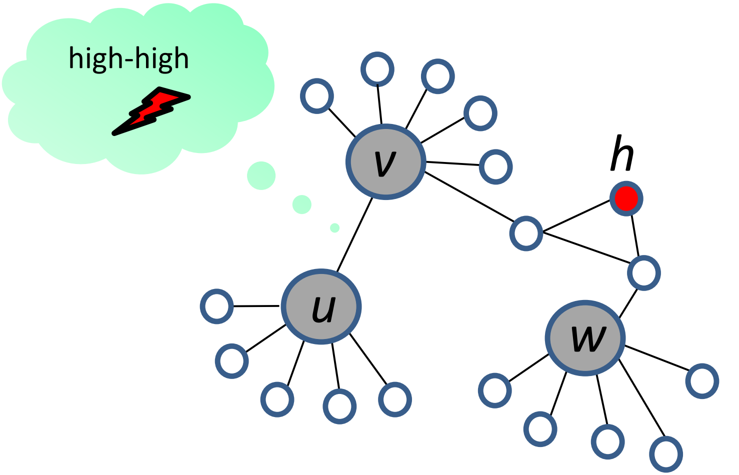
$\Omega(H_\Delta(Y|X))$

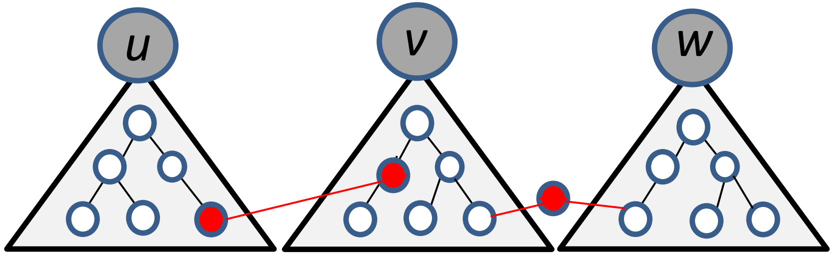
	1	2	3	4	5	6	7
1	0	$\frac{2}{65}$	$\frac{1}{13}$	$\frac{1}{65}$	$\frac{1}{65}$	$\frac{2}{65}$	$\frac{3}{65}$
2	$\frac{2}{65}$	0	$\frac{1}{65}$	0	0	0	$\frac{2}{65}$
3	$\frac{1}{13}$	$\frac{1}{65}$	0	$\frac{2}{65}$	0	0	$\frac{1}{13}$
4	$\frac{1}{65}$	0	$\frac{2}{65}$	0	$\frac{4}{65}$	0	0
5	$\frac{1}{65}$	0	$\frac{3}{65}$	$\frac{4}{65}$	0	0	0
6	$\frac{2}{65}$	0	0	0	0	0	$\frac{3}{65}$
7	$\frac{3}{65}$	$\frac{2}{65}$	$\frac{1}{13}$	0	0	$\frac{3}{65}$	0

$\mathcal{D}$

(Tight) Upper Bounds: Algorithm Idea

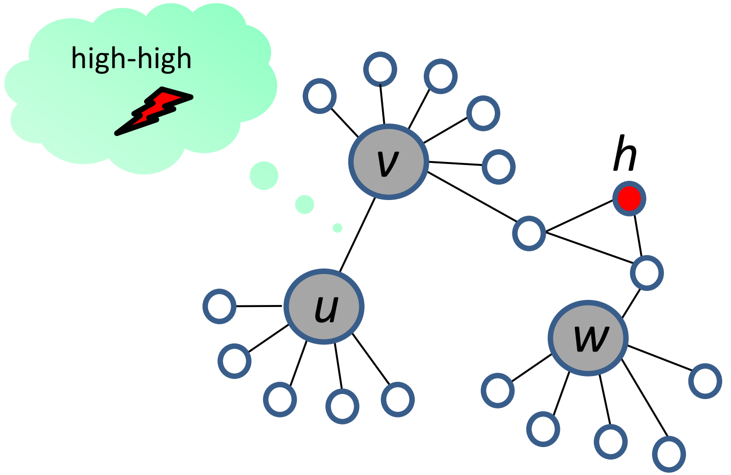
- Idea: construct **per-node optimal tree**
  - BST (e.g., Mehlhorn)
  - Huffman tree
  - Splay tree (!)
- Take **union** of trees but reduce degree
  - E.g., in sparse distribution: leverage **helper** nodes between two "large" (i.e., high-degree) nodes

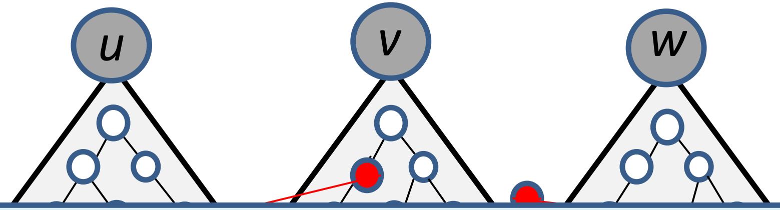




(Tight) Upper Bounds: Algorithm Idea

- Idea: construct **per-node optimal tree**
  - BST (e.g., Mehlhorn)
  - Huffman tree
  - Splay tree (!)
- Take **union** of trees but reduce degree
  - E.g., in sparse distribution: leverage **helper** nodes between two "large" (i.e., high-degree) nodes





Further reading:
IEEE/ACM Trans. Netw. 2016, DISC 2016, DISC 2017

Many Open Questions

- Demand-aware bounded doubling dimension graphs?
- Demand-aware continuous-discrete graphs?
  - Shannon-Fano-Elias coding
- Demand-aware skip graphs?
- …

Conclusion & Future Work

- Communication networks are ***mission-critical but complex***: need for **automated** verification (and **synthesis**: future work)

- Network verification can be fast: **automata-theoretic** approach for MPLS and SR networks runs ***in polynomial-time***

Thank You!

Further Reading

Polynomial-Time What-If Analysis for Prefix-Manipulating MPLS Networks
Stefan Schmid and Jiri Srba.
37th IEEE Conference on Computer Communications (**INFOCOM**), Honolulu, Hawaii, USA, April 2018.

WNetKAT: A Weighted SDN Programming and Verification Language
Kim G. Larsen, Stefan Schmid, and Bingtian Xue.
20th International Conference on Principles of Distributed Systems (**OPODIS**), Madrid, Spain, December 2016.

TI-MFA: Keep Calm and Reroute Segments Fast
Klaus-Tycho Foerster, Mahmoud Parham, Marco Chiesa, and Stefan Schmid.
IEEE Global Internet Symposium (**GI**), Honolulu, Hawaii, USA, April 2018.

Local Fast Failover Routing With Low Stretch
Klaus-Tycho Foerster, Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Tredan.
ACM SIGCOMM Computer Communication Review (**CCR**), 2018.